Courant Computer Science Report #17

September 1979

A Design for Optimizations of the Bitvectoring Class

J. T. Schwartz and M. Sharir

Courant Institute of
Mathematical Sciences

Computer Science Department

New York University

Report No. NSO-17 prepared under Grant No. MCS-76-00116 with the National Science Foundation and Contract No. EY-76-C-02-3077 with the U.S. Department of Energy.

COURANT COMPUTER SCIENCE PUBLICATIONS

COURANT COMPUTER SCIENCE NOTES

A101 ABRAHAMS, P. The PL/I Programming Language, 1979, 151 p.

C66 COCKE, J. & SCHWARTZ, J. Programming Languages and Their Compilers, 1970, 767 p.

D86 DAVIS, M. Computability, 1974, 248 p.

M72 MANACHER, G. ESPL: A Low-Level Language in the Style of Algol, 1971, 496 p.

M81 MULLISH, H. & GOLDSTEIN, M. A SETLB Primer, 1973, 201 p.

S91 SCHWARTZ, J. On Programming: An Interim Report on the SETL Project. Generalities; The SETL Language and Examples of Its Use. 1975, 675 p.

S99 SHAW, P. GYVE — A Programming Language for Protection and Control in a Concurrent Processing Environment, 1978, 668 p.

S100 SHAW, P. " Vol. 2, 1979, 600 p.

W78 WHITEHEAD, E.G., Jr. Combinatorial Algorithms, 1973, 104 p.

COURANT COMPUTER SCIENCE REPORTS

1 WARREN, H. Jr. ASL: A Proposed Variant of SETL, 1973, 326 p.

2 HOBBS, J. R. A Metalanguage for Expressing Grammatical Restrictions in Nodal Spans Parsing of Natural Language, 1974, 266 p.

3 TENENBAUM, A. Type Determination for Very High Level Languages, 1974, 171 p.

4 OWENS, P. A Comprehensive Survey of Parsing Algorithms for Programming Languages, ... 652+ p.

5 GEWIRTZ, W. Investigations in the Theory of Descriptive Complexity, 1974, 60 p.

6 MARKSTEIN, P. Operating System Specification Using Very High Level Dictions, 1975, 152 p.

7 GRISHMAN, R. (ed.) Directions in Artificial Intelligence: Natural Language Processing, 1975, 107 p.

8 GRISHMAN, R. A Survey of Syntactic Analysis Procedures for Natural Language, 1975, 94 p.

9 WEIMAN, CARL Scene Analysis: A Survey, 1975, 62 p.

10 RUBIN, N. A Hierarchical Technique for Mechanical Theorem Proving and Its Application to Programming Language Design, 1975, 172 p.

11 HOBBS, J.R. & ROSENSCHEIN, S.J. Making Computational Sense of Montague's Intensional Logic, 1977, 41 p.

12 DAVIS, M. & SCHWARTZ, J. Correct-Program Technology/Extensibility of Verifiers, with an Appendix by E. Deak, 1977, 146 p.

13 SEMENIUK, C. Groups with Solvable Word Problems, 1979, 77 p.

14 FABRI, J. Automatic Storage Optimization, 1979, 159 p. ..

15. LIU, S-C. & PAIGE, R. Data Structure Choice/Formal Differentiation. Two Papers on Very High Level Program Optimization, 1979, 658 p.

16 GOLDBERG, A. T. On the Complexity of the Satisfiability Problem, 1979, 85 p.

17 SCHWARTZ, J.T. & SHARIR, M. A Design for Optimizations of the Bitvectoring Class, 1979, 71 p.

Notes: Available from Department LN. Prices on request.
Reports: Available from Ms. Lenora Greene. Nos. 1,3,4,6,7,8,10 available in xerox only..

COURANT INSTITUTE OF MATHEMATICAL SCIENCES
251 Mercer Street
New York, New York 10012

COURANT INSTITUTE OF MATHEMATICAL SCIENCES

Computer Science NSO-17

A DESIGN FOR OPTIMIZATIONS OF THE BITVECTORING CLASS

J. T. Schwartz and M. Sharir

Report No. NSO-17 prepared under
Grant No. MCS-76-00116 with the
National Science Foundation and
Contract No. EY-76-C-02-3077 with
the U. S. Department of Energy.

Table of Contents

1. Introduction

This paper describes several general-purpose data-flow analysis algorithms that have been designed and implemented for the SETL optimizer. These algorithms include interval analysis, interprocedural and intraprocedural 'forward' and 'backward' data-flow analysis for 'bitvectoring' problems and code motion. Most of these algorithms use new techniques which improve their performance significantly as compared with traditional methods.

Although these algorithms reflect the special SETL semantic environment, they do so only to a limited extent, and can therefore support a variety of optimizations for most programming languages and systems. In the SETL optimizer they support about half a dozen optimizations, including classical ones, such as redundant expressions elimination, live variables analysis, and reaching definitions analysis, and also including some special SETL optimizations, such as copy elimination and copy motion and elimination of data-conversions.

Our algorithms operate on an intermediate-level representation of the program to be analysed, in which code is partitioned into basic blocks organized as a flow-graph (see next section for a detailed summary of terminology and notations). In preparing to apply these algorithms, we first perform a simple analysis of the interprocedural call pattern of the program to be optimized. This builds up data-structures which are used later in interprocedural data-flow analysis. Then the interval structure of each subprocedure is analyzed, using a variant of an algorithm of Tarjan, which produces a rather compact interval representation. Our variant of Tarjan's algorithm also handles irreducible flow graphs in a simple and efficient manner, and prepares for later code motion algorithms.

After these preliminary analyses have been carried out, four interval-based data-flow algorithms for problems of the 'bitvectoring' type become applicable.

These algorithms solve 'forward' problems (such as available expressions analysis), and 'backward' problems (such as live variables analysis), either intraprocedurally or interprocedurally. The 'forward'algorithms also support code motion. The structure of the interprocedural algorithms reflects the call-by-value semantics of SETL, and would therefore have to be modified for languages allowing parameters passed by reference. However, our interprocedural approach is simple, efficient, and yields sharp results. Attractive bounds on its performance can be proved.

This paper is organized as follows: Section 2 introduces relevant notations and terminology. Section 3 describes our interval analysis algorithm and intraprocedural 'forward' data-flow algorithm. Section 4 presents the interprocedural 'forward' data-flow algorithm and analyzes its performance. 'Backward' data-flow algorithms are discussed in section 5 (in the intraprocedural case) and 6 (in the interprocedural case). Section 7 describes a code motion algorithm as an extension of the forward data-flow algorithms described earlier. Section 8 discusses some possible extensions of bitvectoring data-flow problems, estimates their complexity and assesses the feasibility of adapting our methods for these extended frameworks. Section 9 describes the application of the general algorithms described in previous sections to specific data-flow problems in the SETL optimizer.

SETL code for the various algorithms described in this paper is given in Appendix A. (The actual SETL optimizer is written in SETL, and this code is extracted from it.)

2. Terminology and Notations

In this section we outline the basic notations and terminology to be used in this paper. More information concerning standard terminology can be found e.g. in [He] and [AU].

The program to be analyzed is assumed to have been translated into intermediate-level code, which is partitioned into extended basic blocks, which are single-entry multi-exit code sequences (containing no internal branches). For purposes of interprocedural analysis, we assume that each procedure call instruction constitutes a single-instruction basic block. Moreover, in the interprocedural case, each procedure p is assumed to have a unique entry block, denoted by r_p, and a unique exit block, denoted by e_p, which is also assumed to be a single-instruction block. Optionally, p may also contain a stop block, denoted by s_p, which terminates execution completely when entered (whereas the exit block returns to the point from which p has been called). We also assume that p's entry block r_p is not contained in any loop within p. We assume that program execution always starts at a unique procedure, called the main program and denoted as main, which is not recursive.

In SETL, procedure parameters are passed by value. Value transmission between actual arguments and formal parameters at a procedure call is assumed to be represented in the intermediate-level code by explicit assignment-like argument-passing instructions which occur before and after each call. In the initial form of our algorithms, we treat these special assignments as ordinary assignments, independent of the relevant procedure call, and so ignore some of the tricky issues connected with parameter-passing, such as recursive stacking and unstacking. (Some ammendments to this approach are discussed in section 4.) This allows us to work with a program model in which procedures are parameterless, and in which procedures communicate only via global variables. As in SETL, we disallow procedure variables.

To begin our analysis, we create a flow-graph G_p for each procedure p in the program being analyzed. G_p is a rooted directed graph whose nodes are the basic blocks of p, whose root is r_p, and whose edges are of the form (m,n) where m,n are basic blocks in p, and either n follows m immediately in the code, or else m contains a branch instruction to (the start of) n. We assume that each node in G_p is reachable from its root. The flow graph G for the whole program is then simply taken to be the union of the flow graphs of all its procedures.

The call graph CG of the program is another rooted directed graph whose nodes are the program's procedures, whose root is the main program main, and whose edges are of the form (p,q) where p,q are procedures and p contains a call to q. Again we assume for simplicity that each node in CG is reachable from its root. Note that CG is acyclic iff the program being analyzed is non-recursive.

For any directed graph G and a depth-first spanning tree T for G, we define the loop-connectedness parameter d = d(G,T) of G with respect to T as the maximal number of back-edges (relative to T) lying along an acyclic path in G. Some properties of this parameter are discussed in [He]; see also [KU].

Our next step is to apply interval analysis. Here, we analyze the loop-structure of each procedure flow graph G_p. An interval I with a given entry node in such a graph is required to be a single-entry strongly-connected set of nodes of G_p, having that node as its entry node. This definition differs in a significant detail from the more standard Allen-Cocke definition of an interval [Al_1] and also from the definition of an interval used by Tarjan [Ta]. We build intervals by a technique due to Tarjan, which detects intervals from innermost to outermost, and reduces each such interval to a new single basic block. As it proceeds, our interval analysis algorithm also classifies each interval as being proper (meaning that it is an interval in the classical sense, i.e. has the property that each internal cycle within it contains the entry node) or else improper,

in which case it is an irreducible subgraph.

The preceding remarks outline our approach to control-flow, and we now turn to consider data-flow. In this paper we consider only data-flow problems of the bitvectoring class, which are the simplest data-flow problems that arise in global program optimization. Such data-flow problems are described by a data-flow framework (L,F) (cf.[He],[AU]), for which there exists a finite set E such that $L = 2^E$ and for each $f \in F$ there exist two subsets $A_f \supset B_f$ of E such that for each $x \in L$

$$f(x) = (A_f \cap x) \cup B_f.$$

Heuristically, elements of L represent (Boolean) attribute values (such as availability of expressions, live status of variables, etc.), to be computed at certain program points, and elements of F represent transformations of these values effected by program execution. The special structure of the maps belonging to F allows each f in F to be represented compactly as an element (A_f, B_f) of $L \times L$, and allows functional application, composition and meet (represented as set intersection, see (c) below) to be performed rapidly, using bit-vector and and or operations.

The following facts are standard:

(a) $f(x \wedge y) = f(x) \wedge f(y)$, for each $x,y \in L$ and $f \in F$ (distributivity);

(b) $f^2 = f$, for each $f \in F$ (idempotency);

(c) For each $f, g \in F$, let $g \circ f$ denote the functional composition of g and f, and $g \wedge f$ denote the functional (pointwise) meet of these maps. Then

$$(A_{g \circ f}, B_{g \circ f}) = (A_g \cap A_f \cup B_g, A_g \cap B_f \cup B_g)$$

$$(A_{g \wedge f}, B_{g \wedge f}) = (A_g \cap A_f, B_g \cap B_f),$$

so that F is closed under these functional operations.

(d) The identity map id on L also belongs to F and we have

$$(A_{id},\ B_{id}) = (E, \emptyset)$$

For the sake of convenience we extend the framework (L, F) by introducing a special new and largest element $\Omega \in L$, denoting an undefined data value, and a new function $f_\Omega \in F$, denoting an undefined flow, which maps L into $\{\Omega\}$. All other functions f are extended so that $f(\Omega) = \Omega$. (interpreting elements of L as predicates on the program states, Ω corresponds to the predicate false; f_Ω describes the effect of executing an 'abort' statement.)

Data-flow analysis problems can be either 'forward' analyses (like available expressions analysis), in which attributes are computed by tracing execution flow in a forward direction, from program (or subprocedure) entry up to any given program point, or 'backward' analyses (like live-variables analysis), in which attributes are computed by tracing execution flow in a backward direction, from program (or subprocedure) exits back to any given program point. Also, each analysis can either be performed intraprocedurally, i.e. separately for each procedure, to gather information about the behavior of its local variables, or interprocedurally. In interprocedural analysis a program is analyzed as a whole, to gather information about global variables and procedure parameters. This paper describes algorithms for all combinations of these classes of data-flow analyses.

Consider first intraprocedural forward analysis. Suppose that we are given a flow-graph G_p of some procedure p. Knowing the semantics of the operations within each basic block, we can associate a data-flow map $f_{(m,n)} \in F$ with each edge $(m,n) \in G_p$. This map describes the change in data as control advances from the start of m, through m, to the start of n. (Note that we assume here that the effect of each procedure call within p is already known, which is the case, e.g., if our analysis

deals only with local variables of p, which are unaffected by such a call.) The data-flow problem we wish to solve can be formulated in terms of these functions. Specifically, for each node n in p, let $x_n \in L$ denote the data-value at entry to the basic block n, and let $x_o \in L$ denote (worst-case) attribute information assumed at entry to p. Then we need to solve the following standard set of data-flow equations:

$$x_{r_p} = x_o$$

$$x_n = \Lambda \{f_{(m,n)}(x_m) : (m,n) \in G_p\} \text{ for each node } n \neq r_p \text{ in } p, \tag{2.1}$$

or, more precisely, to compute the maximal fixpoint of these equations.

A variety of algorithms for obtaining this fixpoint are known (cf. [He], [AU] for a survey). In this paper we use an interval-based elimination algorithm, which is described in section 3, and which resembles other such algorithms (such as in [AC], for example), but differs from them in significant details such as the way in which irreducible flow graphs are handled.

Similar equations which relate data at each node to data at its successors rather than its predecessors can be used to describe intraprocedural backward analysis. However, certain significant differences between backward and forward analysis require careful formulation and treatment. Section 5 discusses these issues and describes an intraprocedural interval-based elimination algorithm for the solution of backward data-flow problems.

The next kind of analysis that we consider is interprocedural forward analysis. Two main difficulties must be overcome in adapting the intraprocedural approach sketched above to the interprocedural case. First, we cannot assign data-flow maps

a priori to edges (m,n), where m is a call block, since initially the effect of call blocks is not known. Also, we have to determine an attribute value x_{r_p} at entry to each procedure p. In the intraprocedural case x_{r_p} is generally chosen to reflect worst-case assumptions concerning data at entry to p, but to retain accuracy in the interprocedural case we will only want to make such assumptions at entry to the main program.

The problems just noted are analyzed systematically from a theoretical point of view in [SP, section 3]. There, the preceding data-flow equations (2.1) are reformulated as equations involving data-flow <u>functions</u> ϕ_n, where, for each node n in a procedure p ϕ_n denotes the effect, on the attributes we wish to calculate, of the advance of control from entry to p to the start of n along all interprocedurally valid and balanced paths (i.e. execution paths in which each procedure call is properly terminated). These equations are

$$\begin{aligned} \phi_{r_p} &= \underline{id} \\ \phi_n &= \Lambda \{h_{(m,n)} \circ \phi_m : (m,n) \in G_p\}, \quad n \neq r_p \text{ in } p \end{aligned} \tag{2.2}$$

where

$$h_{(m,n)} = \begin{cases} f_{(m,n)}, & \text{if m is not a call block} \\ \phi_{e_q}, & \text{if m is a call block calling procedure q.} \end{cases}$$

It is shown in [SP] that a (maximal fixpoint) solution of these equations exists if L is finite (which is the case for bitvectoring frameworks), and can be found by methods similar to those available in the intraprocedural case. It is also shown there that this solution coincides with a 'meet over all paths' solution. [SP] discusses an <u>iterative</u> technique for finding that solution; in section 4 we will present a

more efficient interval-based elimination technique. It should be emphasized again that elimination techniques are most appropriate for bitvectoring frameworks, in which functional compositions and meets are almost as easy and fast to perform as functional applications.

Once the solution of (2.2) has been found, we can compute data at procedure entries by solving the following data-flow equations (where x_{r_p} denotes data at entry to the procedure p, and where $x_o \in L$ denotes attributes to be assumed at the start of execution; see [SP, Equations 3.3]):

$$\begin{aligned} x_{r_{\underline{main}}} &= x_o \\ x_{r_q} &= \Lambda \{\phi_c (x_{r_p}) : c \text{ is a call to } q \text{ from } p\} \end{aligned} \tag{2.3}$$

An iterative solution technique for these equations is discussed in section 4. Finally, applying the maps ϕ_n to the entry data x_{r_p} yields data at entry to any program basic block.

A similar approach can be used for interprocedural backward analysis. Specifically, for each basic block n let ψ_n denote the data-flow map describing the effect on the attributes that we wish to compute of the advance of control from the start of n to the exit e_p of the procedure p containing n, along execution paths in which each procedure call is properly terminated, but including also incomplete paths which terminate at some stop block. These maps satisfy the following equations (where $x_o \in L$ denotes worst-case attributes assumed at program exits):

$$\begin{aligned} \psi_{s_p} &= x_o \quad \text{(a constant map), for each p ;} \\ \psi_{e_p} &= \underline{id}_L, \quad \text{for each p ;} \\ \psi_m &= \Lambda \{h_{(m,n)} \circ \psi_n : (m,n) \in G_p\}, \text{ for all other m in p ;} \end{aligned} \tag{2.4}$$

where

$$h_{(m,n)} = \begin{cases} f_{(m,n)}, & \text{if } m \text{ is not a call block} \\ \psi_{r_q}, & \text{if } m \text{ is a call block calling procedure } q \text{ .} \end{cases}$$

Solution methods for these equations, as well as formulation and solution methods for equations analogous to (2.3), are discussed in section 6.

3. A Modified Interval Analysis Technique

In this section we sketch a modified approach to interval analysis, based primarily on Tarjan's interval analysis technique [Ta], but pragmatically adapted. This approach also handles irreducible flow graphs in a reasonably simple and efficient manner.

The classic interval analysis technique of Allen and Cocke (cf. [AC]), builds up a sequence of derived graphs for a given flow graph. Each such graph results from the previous one by simultaneously reducing all first-order intervals to single nodes. This has various disadvantages, e.g. the nodes in one derived graph, even if unaffected by the reduction of this graph, are duplicated in the next derived graph.

Moreover, in this traditional approach, intervals are not required to be strongly connected, which creates extra complications for code motion, where normally we only wish to move code lying in the strongly-connected part of an interval I out of I.

A third potential disadvantage lies in the handling of irreducible flow graphs, which requires special 'node-splitting' mechanisms (see [He]).

A different and potentially more efficient approach to interval analysis which relieves some of these objections,

has been suggested by Tarjan [Ta], and will be followed and adapted here. The relevant theory is suppressed in our description. It can be found in [Ta]; additional exposition of this theory can be found in [SS], whose notation we shall closely follow. A formal SETL code for our interval analysis algorithm is given in Appendix A.

Let $G = G_p$ be a given (intraprocedural) flow-graph having an entry node r, and let T be a depth-first spanning tree for G. We assume for simplicity that r is not a target of a back edge with respect to T. Among other objects, our algorithm will compute a map 'intof' which maps each node in a (strongly-connected) interval to the interval itself. Each interval will be represented by a new flow-graph node, logically placed in G and T 'just before' the first node of the interval (see below for more details). A crude sketch of our interval analysis algorithm is as follows:

(1) Initialize intof to the null map. Mark all nodes as 'proper' (meaning, heuristically, that they are not (as yet) heads of multiple-entry loops). Also initialize a set 'proper-ints' to the null set and a tuple 'intervals' to the null tuple.

(2) Iterate in reverse preorder (of the tree T) through all nodes which are back-edge targets.

(3) For each such node x compute the set

reachunder(x) = {intof$^{\infty}$ (y) : x can be reached from y along a path not going through x whose final edge is a back edge}

where intof$^{\infty}$ (y) = z if $\exists$ k $\geq$ 0 such that intof^{k}(y) = z and intof(z) is undefined.

(4) If the root node r belongs to reachunder(x), then x belongs to a multiple entry loop. Mark x as 'improper' and return to step (2) to process the next back-edge target.

(5) If $r \notin$ reachunder(x), then x is the head of the single-entry loop I(x) = reachunder(x) $\cup\{x\}$. If reachunder(x) contains no improper nodes, then I(x) is the maximal strongly-connected interval (in the classical sense) with head x. Otherwise I(x) is a single-entry irreducible flow subgraph.

(6) Having computed I(x) in (5) we reduce it to a single new node, a representative of which is inserted at a place 'just before' x in G and T. We set intof(y) := I(x) for all $y \in I(x)$. New edges resulting from this graph modification are classified into three categories: real edges, of the form (u, I(x)), which replace edges $(u,x) \in G$, where $u \notin I(x)$; an additional real edge (I(x),x); and virtual edges of the form (I(x),v), where $\exists$ $u \in I(x)$ such that $(u,v) \in G$ and $v \notin I(x)$. After its formation I(x) is added to a set 'intervals' and is placed in a set 'proper-ints' iff it is a proper (reducible) interval. Once this is done, we return to step (2) to process the next back-edge target.

(7) When the iteration at step (2) terminates, all remaining nodes, i.e. nodes x for which intof(x) is still undefined, are nodes not contained within any single-entry loop. Let G' denote the graph resulting from all the reductions that have been carried out. If any node in G' is marked 'improper' then G'is an irreducible flow graph; if not then G'is a DAG (directed acyclic graph). In either case we set intof(y) = r for all $y \in G'$ (i.e. regard G'as an interval, logically identified with its head, the entry node r). Here no modified or additional edges need be created. G'(r) is added to 'intervals' (and will be refered to as the outermost interval). If acyclic, G'is also placed in 'proper-ints'.

Finally, we walk the tree T again, in reverse postorder, to construct a map 'int-nodes', which sends each interval I (proper or improper) to the tuple of all its nodes in reverse post-order, which constitutes an interval order among nodes of proper intervals. The output of our algorithm thus consists

of the following objects:

intof - maps each node to the interval containing it,
intervals - the sequence of all intervals in reverse preorder. (Note that this order is inner-to-outer; i.e. in this order each interval precedes all intervals containing it.)
int-nodes - maps each interval to the sequence of all its nodes in reverse postorder,
proper-ints - the set of all proper (reducible) intervals,
vedges - the set of all additional virtual edges, representing flow in some derived graph of the given flow graph.

Remarks: (1) Tarjan's original algorithm makes no distinction between intervals and their heads. We have chosen to make this distinction for two main reasons: (a) If x is an interval head, and there exists an edge $(x,u) \in G$ such that $u \notin I(x)$, we wish to distinguish between the flow from x to u effected just by that edge, and the flow from x through I(x), to u. It is convenient to do so by introducing I(x) as a separate flow-graph node, which we shall sometimes call the preheader of x. (b) In applying code motion, the entry to an interval I(x) becomes a program point logically different from the entry to its head x, since code will be moved out of the interval loop and inserted between these two points, which makes the above distinction essential. For these reasons we represent intervals as additional flow-graph nodes.

Our treatment of irreducible flow graphs has the following useful features: (a) Irreducibilities are 'localized'. That is, if they are contained within some single-entry loop I, then their effects need be considered only within I. This still allows us to move code out of I. (The same idea of localizing 'bad' flow also plays a role in Rosen's data flow analysis technique [Ro_1].) (b) In any subsequent data-flow analysis step, single entry loops I containing irreducibilities must be analyzed

using iterative techniques. The reverse postorder of nodes serves this purpose quite well. Indeed, according to Hecht and Ullman [HU], no more than d + 1 iterations through the nodes of I are needed for iterative calculation of the relevant flow maps to stabilize, where d is the loop-connectedness parameter of I (cf. [HU], for more details; similar arguments are used in section 4 below to estimate the efficiency of our interprocedural analysis techniques).

Next we describe the application of the data-structures computed by the above interval analysis in solving data-flow problems of the bit-vectoring class. In this section we consider only intraprocedural forward data-flow analysis. The corresponding interprocedural and/or 'backward' analyses are studied in detail in the following sections. Much of the following description is standard (cf. for example [AC]); the novel features of our algorithm mainly reflect our somewhat nonstandard representation of intervals.

We assume that we are given an intraprocedural flow graph G_p, modified by the above interval analysis, plus the various output objects computed by that analysis. In addition, we assume that a preliminary pass through the procedure p (or program) being analyzed has computed a data-flow map $f_{(m,n)} \in F$ as described in section 2 for each <u>non-virtual</u> edge $(m,n) \in G_p$. Let $x_o \in L$ denote the information to be assumed at the procedure entry point.

Data-flow analysis then consists of the following phases.

(1) Elimination Phase

In this phase we iterate through the intervals of the procedure in their reverse preorder (i.e. from inner to outer). In processing an interval I we compute two kinds of new data-flow maps: (a) For each outgoing virtual edge (I,v) we compute a map $f_{(I,v)}$ representing the data-flow from the entry to I (i.e. the entry to its preheader), through I, to the entry to the successor node v. (b) For each node $u \in I$ we compute an auxiliary data-flow map, denoted as $\hat{f}_u$, which represents the data-flow from the entry to the head x of I, along paths contained in I, to the entry to u.

To compute these maps, we iterate through the nodes of I in their interval order (i.e. their relative reverse postorder, or the order in which they appear in int_nodes(I)). For each node u visited in this manner, we apply the following formula:

$$\begin{aligned} \hat{f}_u &= \Lambda \{ f_{(w,u)} \circ \hat{f}_w : (w,u) \in G_p \text{ and } w \in I \}, \text{ if } u \neq x, \\ \hat{f}_x &= \underline{id} \wedge (\Lambda \{ f_{(w,x)} \circ \hat{f}_w : (w,x) \in G_p \text{ and } w \in I \}). \end{aligned} \tag{3.1}$$

Note that the condition $w \in I$ is required to make sure that the edge (w,u) (or (w,x)) is an internal edge of I. For example, w may be an inner interval, so that the edge (w,u) is a virtual edge, resulting from a real edge (w',u), where w' is a node within w. In this case u has two predecessors w, w', and the test $w \in I$ selects only the first one, as desired.

If I is a proper interval, then it is well known that two iterations through the nodes of I are sufficient for the auxiliary maps $\{\hat{f}_u : u \in I\}$ to stabilize, and one iteration is sufficient for the outermost interval (since it is acyclic in this case). If I is improper, then we iterate till information stabilizes and test explicitly for convergence. However, by [HU], the number of iterations required to reach convergence is bounded

by $d_I + 1$, where d_I is the loop-connectedness parameter of I. Moreover d_I is obviously bounded by the number of targets of back edges in I, which,by steps (4)-(6) of our interval analysis algorithm, is equal,if I is an inner interval, to 1 + the number of heads of multiple-entry loops within I but not within a subinterval of I . If we denote this number by M, then at most M + 2 iterations through the nodes of I are required for the stabilization of the above maps. Heuristically, this implies that, in the worst case, not more than one additional iteration is required for each 'source' of irreducibility within I. The same argument obviously also applies to the outermost interval.

After computing the auxiliary maps $\hat{f}_u$ in the above manner, we compute the data-flow map $f_{(I,v)}$ defined above for each virtual successor v of I,using the formula

$$(3.2) \qquad f_{(I,v)} = \Lambda \{f_{(u,v)} \circ \hat{f}_u \circ f_{(I,x)} : u \in I \text{ and } (u,v) \in G_p\}$$

Note that we could also define and compute the auxiliary maps $\hat{f}_u$ in a way which includes the flow through the preheader of I, which would simplify the above formula slightly. Our main reason for not doing so is that we may wish to perform code motion into the preheader of I, and that such motion does not affect the maps $\hat{f}_u$ as we have defined them, but would change them if they also reflected flow through the preheader. We refer the reader to section 7, in which these issues are discussed in greater detail. However, if code motion is not integrated with the analysis which we are now describing, then we can as well include the flow through the preheader in the computation of $\hat{f}_u$, which allows us to replace <u>id</u> by $f_{(I,x)}$ in Equations (3.1), and to rewrite Equations (3.2) as

$$(3.2') \qquad f_{(I,v)} = \Lambda \{f_{(u,v)} \circ \hat{f}_u : u \in I \text{ and } (u,v) \in G_p\}$$

(2) Propagation Phase

In this phase we iterate through the intervals of the procedure in their preorder (from outer-to-inner), propagating data from interval entries to interval nodes, using the auxiliary maps $\hat{f}$ computed in the previous phase.

We begin by initializing the solution map x of our data-flow problem; this is done by putting $x_{r_p} = x_o$. (Here r_p is the procedure entry, and also represents the 'outermost interval' of the procedure.) Whenever we come to process an interval I, we will already have computed attribute data x_I at its entry. If h denotes the head of I, then $\hat{x}_I = f_{(I,h)}(x_I)$ represents attribute data known at entry to the loop of I. Hence, for each $u \in I$, $x_u = \hat{f}_u(\hat{x}_I)$ is the data attribute state at the entry to u. Proceeding in this manner we compute the value of x at entry to each basic block, which completes the intraprocedural solution of the data-flow problem. (If the flow through the preheader of I is already recorded in the auxiliary maps $\hat{f}_u$, then there is no need to compute $\hat{x}_I$, and we have $x_u = \hat{f}_u(x_I)$ for each node $u \in I$.)

It is useful to estimate the time complexity of the above algorithm, which turns out to be rather favorable.

We define the following quantities:

N_{blocks} = number of basic blocks in the flow graph

N_{ints} = number of intervals

N_{irred} = number of multiple-entry loops

$E_{virtual}$ = number of virtual edges

E_{in} = number of internal edges in all intervals (i.e. edges whose source and target belong to the same interval)

E_{out} = number of edges going out of an interval.

Also, for each interval I, we put

N_I = number of nodes in I

M_I = number of multiple-entry loops in I

E_I = number of internal edges in I.

We can then compute the time required by the above algorithm as follows: Suppose that each bit-vector operation (and, or) takes one unit of time. Then it is easily seen (cf. section 2 for details) that functional application takes 2 time units; functional meet takes 2 time units, and functional composition takes 4 time units. If we assume that the analysis to be performed involves no code motion, then the application of the modified Equations(3.1) will require at most

$$\sum_I (6\ E_I - 2\ N_I + 2)\ (M_I + 2 - \delta_{Ir})$$

time units, where the summation is over all intervals I (note that the number of elementary meet operations required to take a meet over a set S is $|S| - 1$). Similarly, Equations (3.2') will require

$$6\ E_{out} - 2\ E_{virtual}$$

time units and the modified propagation phase will require

$$\sum_I 2\ N_I = 2\ (N_{blocks} + N_{ints} - 1) \text{ time units.}$$

For example, if the procedure flow graph is reducible, then the total number of time units required by the algorithm is at most

$$12\ E_{in} - 4(N_{blocks} + N_{ints} - 1) + 4\ N_{ints} - 6\ E_r + 2\ N_r - 2$$
$$+ 6\ E_{out} - 2\ E_{virtual} + 2\ (N_{blocks} + N_{ints} - 1)$$
$$= 12\ E_{in} + 6\ E_{out} - 2\ E_{virtual} - 6\ E_r - 2\ N_{blocks} + 2\ N_{ints} + 2\ N_r$$

4. Interprocedural Forward Data-flow Analysis

In this section we describe our technique for interprocedural forward data-flow analysis. We will also analyze its performance and efficiency in terms of the call graph structure of the program being analyzed, and will discuss various possible improvements and extensions of our approach.

We remind the reader that our model of a procedural program is one in which procedures are parameterless (or more generally, in which parameters are called by value). Thus the aim of our interprocedural analysis is to determine the properties of global variables. The model we use evades certain difficult problems such as analysis of 'aliasing' that would arise in the presence of reference parameters (cf. [Ro_2]), but in the semantic environment which we assume our algorithm yields sharp interprocedural data-flow information.

The algorithm to be presented below is based on a prior study of interprocedural analysis by Sharir & Pnueli [SP], and is in fact merely a simple and efficient implementation of the 'functional approach' described in [SP] (see also section 2). However, we justify this algorithm by some new theoretical results concerning interprocedural flow, which are detailed below.

Our interprocedural algorithm is quite similar to the intraprocedural data-flow algorithm given in the previous section. The description given below will emphasize the modifications and extensions of the previous algorithm needed for interprocedural analysis.

In general terms, the interprocedural algorithm consists of the following phases:

(a) Initialization. In this phase we compute data-propagation maps $f_{(m,n)}$ for all non virtual edges $(m,n) \in G$ such that m is not a call block. If m is a call block then the effect of flow through m is not known *a priori*, and the determination

of that effect is in fact one of the major goals of our analysis; for such m, $f_{(m,n)}$ is initially left undefined (or rather is set to f_Ω to indicate undefined flow).

(b) Call-graph Analysis. This phase is independent of the particular interprocedural bit-vector data-flow problem in question, and so should be performed once prior to any solution of such a problem. In this phase we analyze the structure of the program's call graph CG, as follows: We first construct a depth-first spanning tree T for CG, then find all strongly-connected components $S_1, S_2, \ldots, S_k$ of CG arranged in reverse postorder (with respect to T) of their roots. We also arrange the nodes within each S_i in their reverse postorder. For this purpose, we use an efficient new algorithm, differing from known algorithms for constructing strongly-connected components of a directed graph (cf. [AHU, section 5.7]) so as to list components in both external and internal reverse postorder. This algorithm is presented in Appendix A. In addition, for each strongly-connected component S_i, we estimate the loop-connectedness parameter $d_i \equiv d(S_i,T)$. Actually, since efficient computation of d_i may not be possible in general, we make do by overestimating it; our estimate for d_i is simply the number of back-edge target nodes (with respect to T) in S_i, which will be denoted as d_i'. Obviously $d_i' \geq d_i$ and $d_i' = d_i$ if S_i consists of a single node. Note that a typical call graph can be expected either to be acyclic (for non-recursive programs), or at worst to contain several simple-loops, each corresponding to some simply-recursive procedure. Thus, for such call graphs each component S_i will indeed consist only of a single node p, and accordingly $d_i = d_i' = 0$ if p is non-recursive, $d_i = d_i' = 1$ if p is recursive.

(c) Elimination Phase: In this phase we iterate through the program procedures in the following order: First we iterate once through the strongly connected components of CG in their

postorder (i.e. in the order S_k, S_{k-1},...,S_1), and for each such component S_i we iterate through its procedures in their relative postorder at most $2\,d_i' + 1$ times. Each procedure p visited in this iteration is subjected to an interval-based elimination pass, quite similar to the elimination phase of the intraprocedural algorithm given in the previous section, but preceded by the following resetting of flow maps for call blocks: For each call block c and its (unique) successor v, we compute the associated data-flow map $f_{(c,v)}$ (which is not yet available from the initialization phase (a)) using the formula

(4.1) $\quad f_{(c,v)} = \phi_{e_q}$, where q is the procedure called by c. Note that if ϕ_{e_q} is still undefined (e.g. if q has not yet been processed), then it is taken to be f_Ω.

While computing these maps, we also do the following two things to speed up the algorithm:

(i) If p is being reprocessed (as will happen if p is recursive), and for each call block c and its successor v in p, $f_{(c,v)}$ has not changed from its value in the last processing of p, then obviously there is no need to process p again, since the analysis results will not have changed. If processing of all procedures in S_i has stabilized in this manner, then the processing of S_i has converged, and we go on to analyze the next component S_{i-1}.

(ii) Even if full convergence as in (i) has not yet been achieved in p, we can still bypass a considerable part of the reprocessing of p by noting that for a given interval I in p, it is pointless to re-analyze I if no call-block flow-maps $f_{(c,v)}$, for c a node in I or in a subinterval of I, have changed since the last processing of p. Skipping the reprocessing of such intervals will speed up repeated processings of a procedure substantially.

Later in this section we will show that the elimination phase computes the correct values of the auxiliary maps $\hat{f}_u$ and the extended-flow maps $f_{(I,v)}$. (Note also that in the interprocedural case these maps will account for flow along interprocedurally valid balanced paths only; compare this to the way in which the maps ϕ_n are defined in section 2.)

Remarks: (1) We do not compute the maps ϕ_n defined in section 2 explicitly; however, they can be easily constructed from the maps actually computed, as will be demonstrated below. Nevertheless, the maps ϕ_{e_q} which are needed in the call-block maps resetting subphase, are actually available, since $\phi_{e_q} = \hat{f}_{e_q}$, because e_q always lies in the outermost interval of the procedure q.

(2) If the program to be analyzed is nonrecursive, then all the procedures are processed just once in their postorder. It is easy to check that this order constitutes an 'inverse invocation order' in the terminology of [Al_2].

(d) Calculation of Data at Procedure Entries. This phase establishes data-values at procedure entries by solving equations (2.3). We convert these equations to data-flow equations for the call graph CG, by defining the map

$$g_{(p,q)} = \Lambda\{\phi_c : c \text{ is a call from p to q}\} \tag{4.2}$$

for each edge $(p,q) \in CG$, so that equations (2.3) reduce to the following equations (where z_p denotes x_{r_p}, the data at entry to p):

$$\begin{aligned} z_{\underline{main}} &= x_o \\ z_p &= \Lambda\{g_{(q,p)}(z_q) : (q,p) \in CG\}, \text{ for each } p \neq \underline{main} \end{aligned} \tag{4.3}$$

However, before attempting to solve these equations, we first have to compute the maps ϕ_c appearing in (4.2), which are not immediately available from phase (c). For this we use the formula

$$\phi_c = \hat{f}_c \circ \hat{f}_{I(c)} \circ \cdots \circ \hat{f}_{I^k(c)} \tag{4.4}$$

where I(c) = intof(c), the interval containing c, and where $I^{k+1}(c)$ is the outermost interval of the procedure p containing c. (Of course, (4.4) is applicable to any block c.) To justify (4.4), we observe that any (interprocedurally valid and balanced) path from r_p to c can be decomposed as the concatenation of paths, the first of which leads from r_p to (the entry of) $I^k(c)$, through nodes of $I^{k+1}(c) = r_p$, the second of which leads from the entry to $I^k(c)$ through nodes of $I^k(c)$ to the entry to $I^{k-1}(c)$ and so on. (Of course, this justification still depends on the (still unproven) assertion that final value of the auxiliary maps $\hat{f}$ computed in phase (c) correctly represent the effect of the corresponding flows; this assertion will be proven later in this section.)

Since in most cases the program's call graph will be quite simple (and in any case its size can be expected to be much smaller than the sizes of the flow graphs analyzed in the previous phase), solution of equations (4.3) will generally be easy. To get this solution, we use the following iterative approach: We iterate once over the strongly-connected components S_i of the call graph in their relative reverse postorder, and for each such S_i we iterate over its procedures in their relative reverse postorder, applying (4.3) till data stabilizes at their entries, but no more than $d_i' + 1$ times. That this number of iterations is sufficient can be shown using arguments similar to those in [HU]. In fact, this iteration technique is applicable to the solution of similar data-flow equations for any flow graph, and is a trivial, but significant improvement of the Hecht-Ullman iteration method [HU]. It is also a special case of Kennedy's node-listing technique [Ke].

(e) Data-propagation. This is the simple final phase of our algorithm. Here, using the entry information provided by phase (d) and the auxiliary maps of phase (c), we propagate data to each node in the program flow graph. This is done precisely as in the propagation phase of the intraprocedural algorithm given in the previous section, only here we begin the propagation by setting $x_{r_p} = z_p$ for each procedure p. This propagation completes our analysis.

We now turn to analyze the performance of the elimination phase (c) of our algorithm, and to justify its correctness. We will investigate the behavior of phase (c) for arbitrary orders of iteration through the program's procedures, and will show that our iteration order is also particularly efficient.

Let us first introduce some notations. Let p be a procedure and n a node in p. Using the notations of [SP], we define $IVP_o(r_p, n)$ as the set of all interprocedurally valid balanced execution paths (i.e. interprocedural paths in which procedure calls and returns are executed in a proper sequence, i.e., each return matches the last uncompleted call and all calls are subsequently completed) leading from r_p to n. For each path $\pi \in IVP_o(r_p, n)$ we define $CS(\pi)$ as the set of all calling sequences materialized during the execution of π, each such calling sequence being the invocation-order sequence of all procedures invoked and not yet completed, as some initial subpath π' of π is executed. Note the obvious one-one correspondence between calling sequences and paths (without their initial node) in the call graph CG.

Suppose now that our elimination phase (c) iterates through the program procedures in some arbitrary order $p_1, p_2, \ldots, p_n, \ldots$ (where repetition is allowed), and attains convergence. The general results of [SP] ensure convergence provided that sufficiently many iterations are applied, because every 'bitvectoring' data-flow framework has a finite semilattice L.

Let p_k be the k-th procedure processed in the assumed order. For any node n in the program, let ϕ_n^k denote the value of ϕ_n (given by (4.4)) immediately after p_k had been processed. It follows by standard arguments that this processing of p_k will yield the fixpoint solution of equations (2.2), for the nodes of p_k, provided that each procedure q called from p_k is assumed to have the data-flow effect described by the map $\phi_{e_q}^{k-1}$.

The significance of calling sequences is seen from the following observation. As shown in [SP, section 3], for each procedure p and each node n in p the final value of the maps ϕ_n satisfies

$$\phi_n = \Lambda\{f_\pi : \pi \in IVP_0(r_p, n)\} \tag{4.5}$$

Let $\pi \in IVP_o(r_p, n)$. At what point in phase (c) can we be sure that f_π has already been included in the meet which defines the current value of ϕ_n? A sufficient condition is given in the following lemma:

Lemma 4.1: Suppose that phase (c) has already processed procedures $p_1, p_2, \ldots, p_k$ in order and is now analyzing $p_{k+1} = p$. If the reverse sequence of each calling sequence in $CS(\pi)$ is a subsequence of $p_1, p_2, \ldots, p_k$, then, at the end of the current processing of p we have $f_\pi \geq \phi_n^{k+1}$.

Proof: It follows from a remark made above that when we have finished processing p we will have

$$\phi_n^{k+1} = \Lambda\{f_\pi^* : \pi \in IVP_o(r_p, n)\} \tag{4.6}$$

where f_π^* is defined as follows: consider π as a path lying wholly in p, paying no attention to the flow within invoked subprocedures. Suppose that when mapped in this way π becomes $(r_p, s_1, s_2, \ldots, s_j, n)$. We then put

(4.7) $$f^*_\pi = h_{(s_j,n)} \circ h_{(s_{j-1},s_j)} \cdots \circ h_{(s_1,s_2)} \circ h_{(r_p,s_1)}$$

where

(4.7') $$h_{(m,n)} = \begin{cases} f_{(m,n)}, & \text{if } m \text{ is not a call block} \\ \phi^k_{e_q}, & \text{if } m \text{ is a call to a procedure } q \end{cases}$$

Our proof now proceeds by induction on k. If $k = 0$, then no π satisfying the assumptions of the lemma can contain procedure calls. For such π we have $f^*_\pi = f_\pi$, and the assertion of the lemma follows immediately from (4.6).

Next suppose that the lemma is true up to and including some $k \geq 0$, and let π be a path satisfying the assumptions of the lemma. Suppose that π, restricted to p, has the form $(r_p, s_1, s_2, \ldots, s_j, n)$ as above. Then the actual path π is equal to $\pi_1 || \pi_2 || \ldots || \pi_{j+1}$, where for each $i \leq j + 1$

$$\pi_i = \begin{cases} (s_{i-1}, s_i), & \text{if } s_{i-1} \text{ is not a call block} \\ \text{the subpath of } \pi \text{ corresponding to the execution of the procedure called at } s_{i-1}, & \text{if } s_{i-1} \text{ is a call block.} \end{cases}$$

By the induction hypothesis we have $h_{(s_{i-1},s_i)} \leq f_{\pi_i}$, $i=1 \ldots j+1$, so that $f^*_\pi \leq f_\pi$; thus $\phi^{k+1}_n \leq f^*_\pi \leq f_\pi$ by (4.6). This proves the lemma. Q.E.D.

One immediate corollary of this lemma is that if the call graph of the program is non-recursive (i.e. acyclic) then it is sufficient to process each procedure once if processing is in inverse invocation order, to ensure stabilization of interprocedural attributes. Indeed, if this order of processing is used then at the time p is processed, all procedures which can be called from p either directly or indirectly will already have been processed. Thus

for each node n in p and each $\pi \in IVP_o(r_p, n)$, π satisties the conditions of the preceeding lemma. Consequently, when finished with processing of p we will have $\phi_n \leq f_\pi$, so that by using (4.5) it follows easily that the value of ϕ_n at the end of processing is already equal to the desired value of this map. This observation is essentially due to Allen [Al_2].

Suppose next that the call graph is recursive (cyclic). In this case paths in the call graph (and hence also calling sequences) can be arbitrarily long, so that it is infeasible to apply lemma 4.1 to all paths π in order to determine when stabilization must occur.

However, in this case we can make use of the common device of looking for a relatively small subset of the set of all relevant paths π which has the property that it is sufficient to trace the flow through these paths to obtain the desired data-flow quantities. Kennedy's node listing algorithm [Ke], Kam and Ullman's study of the Hecht-Ullman algorithm [KU] and Sharir and Pnueli's study of another interprocedural data-flow technique [SP, section 5] all employ this technique.

Using the special properties of bit-vector data-flow frameworks, we will now show the existence of an appropriate 'exhaustive' subset of paths. Each bit-vector data-flow framework (L,F) is <u>1-related</u> in the terminology of [SP,section 5], that is, (suppressing the extension of L by Ω) L can be decomposed as $\{0,1\}^E$ for some (finite) set E,and each $f \in F$ admits a decomposition $(f^\alpha)_{\alpha \in E}$ such that for each $x \in L$, $(f(x))_\alpha = f^\alpha(x_\alpha)$, and each f_α is either constant (0 or 1) or the identity <u>id</u> on {0,1}. This implies that for each execution path $\pi = (n_1, n_2 \ldots, n_k)$ and each $\alpha \in E$ either $f^\alpha_{(n_i, n_{i+1})} = \underline{id}$ for all $i < k$, or else there exists an index $s < k$ such that $f^\alpha_{(n_s, n_{s+1})}$ is constant and $f^\alpha_{(n_i, n_{i+1})} = \underline{id}$ for all $i > s$. In the first case $f^\alpha_\pi = \underline{id}$ and in the second $f^\alpha_\pi = f^\alpha_{(n_s, n_{s+1})}$. Thus f^α_π depends on <u>at most</u> one point along π.

It is helpful to attach the following heuristic interpretation to the above observation: Consider some specific component $\alpha \in E$ of our bit vector. Then each flow effect can be considered to be either an 'event' (corresponding to $f^{\alpha} = 1$), or an 'anti-event' ($f^{\alpha} = 0$); in the absence of any event or anti-event, we have 'transparency' ($f^{\alpha} = \underline{id}$). For example, in available expressions analysis, each $\alpha \in E$ is an expression whose availability is to be analyzed. An 'event' is a generation of α, an 'anti-event' is a kill of α, and a transparent flow is one in which α is neither generated nor killed. The flow effect of a path π thus depends on the last non-transparent edge effect in π (if any); if this is an event, then π is said to <u>create an event</u>, and if this is an anti-event, π is said to <u>create an anti-event</u>; otherwise π is <u>transparent</u>. In forward-intersection data flow problems, we wish to determine for each node n whether <u>all paths</u> leading to n create an event, whereas in forward-union problems (where lattice-meet is set union rather than intersection), we wish to determine whether there exists <u>at least one</u> event-creating path leading to n.

<u>Lemma 4.2</u>: Let (L,F) be a bit vector data-flow framework as above. Then for each $\alpha \in E$, procedure p, node n in p and path $\pi \in IVP_o(r_p, n)$ there exists another path $\pi' \in IVP_o(r_p, n)$ (which passes only through nodes of π) such that

(i) $f^{\alpha}_{\pi'} = f^{\alpha}_{\pi}$

(ii) Every calling sequence in $CS(\pi')$ is the concatenation of two sequences, neither of which contains any procedure more than once.

<u>Proof</u>: We can assume that π does not satisfy (ii) for otherwise we can simply take $\pi' = \pi$. Let us first describe our proof heuristically. Suppose that π is event-creating, and that this event occurs in some node m (possibly in another procedure).

Then, instead of tracing the flow of π we can go (through nodes of π) in the shortest possible way to m and then in the shortest possible way back to n. However, the fact that we insist on interprocedural validity may add extra constraints on the above path-shortening process. (Note that in intraprocedural analysis the absence of such constraints simplifies the situation considerably. For example, the initial part of the new path need not even trace nodes of π, but can be any shortest acyclic path from r_p to m. Moreover, interprocedurally the terminal part of the new path must consist of nodes of π, to ensure transparency along the path after the event has occurred, but in the intraprocedural case can be chosen to be acyclic.)

To make this idea more precise, let $\pi = (r_p = n_1, n_2, n_3, \ldots, n_k, n)$ and let $s \leq k$ be the largest index such that $f^{\alpha}_{(n_s, n_{s+1})}$ is constant (the last event or anti-event along π). If there is no such index, s is undefined. Suppose that some calling sequence corresponding to some initial subpath π_1 of π contains a procedure q twice. Then we can shorten π as follows. π_1 contains two different calls c_1, c_2 to q where neither call is completed in π_1 (though both are completed in π later on). Let π' be the path obtained if we execute π up to c_1, then execute q in the same way that π does from the <u>second</u> time q is invoked till the corresponding return, but then return to the block following c_1 instead of returning to the block following c_2; and finally follow π from there as if the <u>first</u> invocation of q has been completed. Obviously $\pi' \in IVP_o(r_p, n)$ and π' is shorter than π. If π transparent, then obviously π' is also transparent and we can keep shortening π' in this manner till (ii) is also satisfied, and, in fact, till all calling sequences in $CS(\pi')$ contain each procedure at most once.

If π is non-transparent, then we can carry out the above shortening process as long as n_s is not deleted (that is, n_s is neither in the subpath of π from c_1 to c_2, nor in the subpath

between the corresponding returns from q). Suppose that we have shortened π in this manner as much as possible to obtain π' which still contains n_s. Obviously $f^{\alpha}_{\pi'} = f^{\alpha}_{\pi}$, and we claim that π' satisfies (ii). Indeed, if not, then there exists a calling sequence materializing during the execution of π' which is not the concatenation of two sequences none of which contains a procedure more than once. Let T be that calling sequence, and let T_1 be the largest initial subsequence of T in which no procedure appears more than once. Then we can write $T = T_1 \,||\, [p] \,||\, T_2$ where p appears also in T_1. Now either p appears also in T_2, or else some other procedure q appears twice in T_2. In the first case we have three calls c_1, c_2, c_3 to p along π', none of which has been completed when the next one is made, and in the second case we have four calls c_1, c_2, c_3, c_4 along π', where c_1, c_2 are calls to p, c_3, c_4 are calls to q, and where none of these calls has been completed when the next one is made. Then it is clear that we can apply the above shortening process to π', using in the first case either the calls c_1, c_2 or the calls c_2, c_3, and in the second case either the calls c_1, c_2 or the calls c_3, c_4, to obtain a shorter path which still contains n_s. This contradicts the definition of π' and it follows therefore that π' satisfies (ii). Q.E.D.

Remark: The above argument is very similar to that used in Lemma 5.3 and theorem 5.6 of [SP].

Corollory 4.3: Let (L,F) be a bitvector data-flow framework. Then, for each procedure p and each node n in p, we have

(4.8) $$\phi_n = \Lambda\{f_\pi : \pi \in IVP_o(r_p, n) \text{ such that each calling sequence in } CS(\pi) \text{ is the concatenation of two sequence none of which contains a procedure more than once}\}$$

Proof: By lemma 4.2, for each $\alpha \in E$ we have

$$\Lambda\{f_\pi^\alpha : \pi \in IVP_o(r_p,n)\} = \Lambda\{f_\pi^\alpha : \pi \text{ as in the right-hand side of (4.9)}\}$$

and the lemma follows immediately from (4.5). Q.E.D.

Lemma 4.1 and corollory 4.3 can now be combined to deduce the following 'node listing' principle:

Definition: A doubled node listing for the call graph of a program is a sequence S of procedures having the property that each path in the call graph which is a concatenation of two acyclic paths (excluding its initial node) is a subsequence of S.

Theorem 4.9: Let S be any doubled node listing for the call graph, and suppose that phase (c) of our basic algorithm is performed by processing procedures in the reverse of their order in S. Then, at the end of one iteration through S, the algorithm will converge and all data-flow maps will have their final desired value.

Proof: Immediate from lemma 4.1 corollory 4.3 and the correspondence between call graph paths and calling sequences. Q.E.D.

Note however, that phase (c) does not compute the maps ϕ_n directly, but instead computes auxiliary maps $\hat{f}_n$ for each flow graph node n. It is a simple matter to extend lemma 4.1 and corollary 4.3 to handle such maps as well, and we leave this as an exercise to the reader. As a by-product of such arguments, one can also prove (4.4) rigorously.

In view of the preceding theorem, analysis of the elimination phase of our algorithm reduces to showing that the iteration order actually used in this phase does constitute a doubled node listing for the program's call graph. The order in which procedures are processed during elimination is the reverse order of the tuple

$$S = \sum_{i=1}^{k} (2d_i' + 1)*S_i \tag{4.9}$$

of procedures where the procedures within each strongly connected component S_i of the call graph are arranged in their reverse tree postorder, and where summation corresponds to tuple concatenation while multiplication by an integer represents tuple replication. To show that this S is indeed a doubled node-listing for the call graph, we proceed as follows:

Lemma 4.5: Let π be a path in CG. Then π can be decomposed as $\pi_{i_1} || \pi_{i_2} \ldots || \pi_{i_m}$ where each π_{i_j} consists only of nodes of S_{i_j} and where $1 \leq i_1 < i_2 \ldots < i_m \leq k$.

Proof: Suppose the contrary. Let $S_{i_1}, S_{i_2} \ldots S_{i_m}$ be the sequence of strongly connected components of CG through which π passes in this order; for each $j \leq m$ let π_{i_j} be the part of π that lies within S_{i_j}. Then for at least one $j < m$ we have $i_j > i_{j+1}$. By the strong connectivity of the components S_i it follows that there exists a path π_0 from the root p of S_{i_j} to the root q of $S_{i_{j+1}}$. Since $i_j > i_{j+1}$, p has a lower postorder index than q in the depth-first spanning tree T which establishes the order of the strong components S_i. Thus either q is an ancestor of p in this tree or else q is to the right of p. In the first case the existence of the path π_o implies that p and q must belong to the same strongly connected component. The second case is impossible since the edge connecting π_{i_j} to $\pi_{i_{j+1}}$ would be a left-to right cross edge. This contradiction proves our lemma.

Q.E.D.

Theorem 4.6: The S of (4.9) above is a doubled node listing for the call graph.

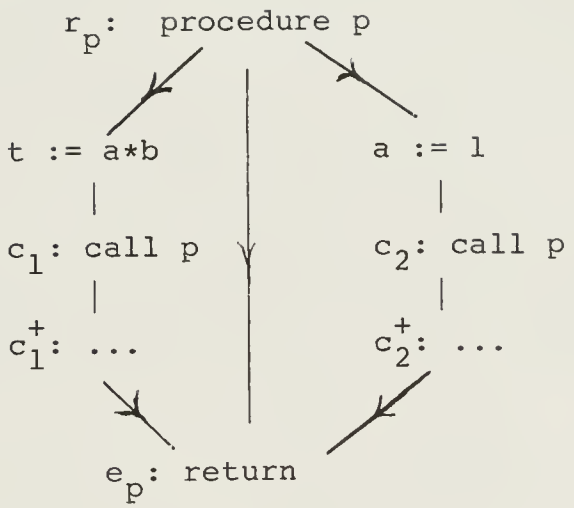
Proof: Let π be a path in the call graph which (ignoring its initial node) is a concatenation of two acyclic paths π' and π''.

Decompose π as in the previous lemma. That lemma implies that it is sufficient to prove that each π_{i_j} is a subsequence of $(2\ d'_{i_j} + 1) * S_{i_j}$. Hence we can assume with no loss of generality that π is a path in S_i for some $i \leq k$. Since both π' and π'' are acyclic, it follows from the definition of d_i that π contains no more than $2\ d_i$ back edges. The nodes of S_i are ordered in S_i in such a way that the only edges from a node to a previously listed node are back edges with respect to T. Hence any part of π between two subsequent back edges is a subsequence of S_i and it follows that π is a subsequence of $(2\ d_i + 1) * S_i$, and therefore also of $(2\ d'_i + 1) * S_i$. Q.E.D.

Remark: This result is analogous to Hecht and Ullman's bound [HU] on the number of iterations required for their (intraprocedural) data-flow algorithm to converge, and also with the bound on iterations through interval nodes given in section 3, and a similar bound for backward problems to be given in the next section. The difference between Hecht and Ullman's bound $(d + 1)$ and our bound $(2\ d + 1)$ reflects additional interprocedural constraints on execution paths that do not allow us to shorten paths as much as is possible in the intraprocedural case.

We shall now show that the bound $2\ d + 1$ cannot be improved in general. Consider a strongly connected component S consisting of a single recursive procedure p. Here $d = 1$ and the above results indicate that p may have to be processed 3 times. Indeed, the following example shows that processing p only twice may fail to produce the desired maps values.

Example 1:

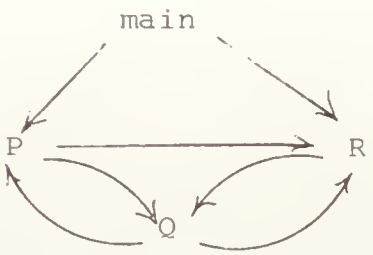


Consider analysis of the availability of the single expression a * b. To detect that $\phi_{c_1^+} = 0$ (i.e. that a * b may be killed during the corresponding flow), we need to trace flow along the path

$$\pi = (r_p, c_1, r_p, c_2, r_p, e_p, c_2^+, e_p, c_1^+)$$

and this can be done only by processing p three times. At the end of a first iteration we will have $\phi_{e_p} = \underline{id}$, and $\phi_{c_1^+} = f_\Omega$ as only the middle path can be traced during that iteration. During the second iteration we will have $\phi_{c_1^+} = \underline{1}$, since the effect of the call c_1 is taken as ϕ_{e_p} as obtained from the last iteration, i.e. $\underline{id}$; but at the end of this iteration we will arrive at the correct value $\phi_{e_p} = \underline{0}$ by propagating through the right-hand path containing c_2. Using this value during the third iteration over p will then yield the correct value for $\phi_{c_1^+}$.

Example 2: Consider the call graph



which has the following depth-first spanning tree

Here there are two strongly connected components

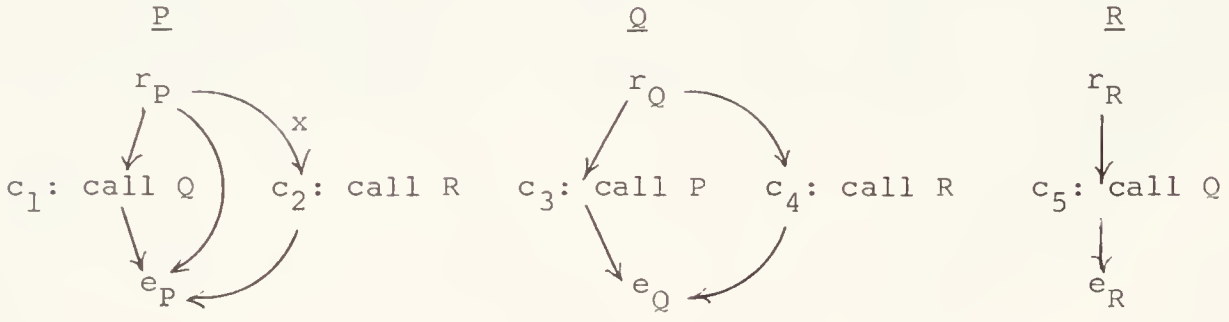
S_1 = [main] $d_1 = 0$

S_2 = [P, Q, R] $d_2 = 2$

so that

S = [main, P,Q,R,P,Q,R,P,Q,R, P,Q,R,P,Q,R]

Suppose that the procedures in question have the following internal form.



where x denotes a kill of a*b, and where no other node in these routines affects a*b. Then to deduce that $\phi_{e_R} = \underline{0}$ we have to trace the following path

$$\pi = (r_R, c_5, r_Q, c_3, r_P, x, c_2, r_R, c_5, r_Q, c_3, r_P, e_P, c_3^+, e_Q, c_5^+, e_R, c_2^+, e_P, c_3^+, e_Q, c_5^+, e_R)$$

during which the calling sequence (Q, P, R, Q, P) materializes. It is easy to see that if we process procedures in the reverse order of S, then we will arrive at the correct value for ϕ_{e_R} only during our fifth iteration over R. Indeed, the following table shows the flow effect through each procedure computed as this procedure is processed:

Z: procedure processed	R	Q	P	R	Q	P	R	Q	P	R	Q	P	R	Q	P
ϕ_{e_z}	f_Ω	f_Ω	$\underline{id}$	f_Ω	$\underline{id}$	$\underline{id}$	$\underline{id}$	$\underline{id}$	$\underline{0}$	$\underline{id}$	$\underline{0}$	$\underline{0}$	$\underline{0}$	$\underline{0}$	$\underline{0}$

(Of course, in this case we do not claim that the above doubled node listing S is optimal; all we claim is that if our standard ordering of strongly-connected components is used, then 5 repetitions of [P,Q,R] is necessary in this case.

An issue yet to be considered is the adaption of our technique to analysis of programs containing procedures with parameters. Assume first that parameters are passed by value (as is the case in SETL). Then only two modifications of our method are required. Till now we have assumedthat the flow effect of a call block c (in regard to the elimination phase (c)) is identical with the flow-effect of the invoked procedure q, and accordingly have set $f_{(c,v)} = \phi_{e_q}$. To allow for the presence

of parameters we must replace ϕ_{e_q} in this assignment by another map which reflects both the binding of actual arguments to formal parameters and the stacking and unstacking of parameters in recursive cases. This new map may not belong to F, so that the performance analysis which we have given for the elimination phase (c) may no longer be valid, and extra iterations may be required to ensure convergence. Moreover, we have assumed that during the entry-data phase (d) data does not change between a point of call to a procedure q and its entry. This need not be true if q has parameters, and thus equations (4.2) and (4.3) have to be modified by replacing each map ϕ_c by another map which also takes the pre-call action at c into account (e.g. the transmission of input arguments to input parameters). These maps may again not be of the bitvectoring type, which means that here too the iteration bounds that we have given may no longer be correct and extra iterations may be required.

Of course, one can compromise by assuming (as we did in section 2) that value-passage between actual arguments and formal parameters at a procedure call c is made explicit in the code by assignment-like statements before and after the call, which are then analyzed as regular assignments independent of c itself. This approach, while safe, can lead to some loss of accuracy, as it ignores some details of parameter passage. Nevertheless, this approach allows the algorithm we have described to be used without modification and the same efficient iteration bounds still apply.

The situation becomes considerably more complicated when parameters passed by reference are allowed. To retain accuracy in this case, our algorithm must be substantially modified, to take the possibility of 'aliasing' (cf. [Ro], [Ba] into account). We have ignored this problem as it does not arise in optimizing SETL. However, it seems likely that a variant of our method, preceded by some kind of aliasing analysis (such as that described in [Ba]) can be applied in this case. However, it will probably be hard to avoid some loss of accuracy.

5. Intraprocedural Backward Data-flow Analysis

In this section we describe an approach to intraprocedural backward data-flow problems which we extend to the interprocedural case in the next section.

In backward data-flow analysis information is propagated in the reverse direction of control flow, from program exits backward. Such an analysis aims to determine at each program point n what might (or must) occur after control has reached n. Backward analysis is used for various purposes, e.g. to compute the live-dead status of program variables ([He],[AU]), and also in conjuction with a forward analysis in order to determine safety and profitability of certain optimizations (see section 9 for a list of the analyses of this kind used in the SETL optimizer).

One can always view backward data-flow analysis as a forward analysis applied to inverse flow-graph. But if this device is used, the relevant program structures, such as the interval structure, may become unfavorable. Indeed, the inverse flow-graph is in general not reducible, and even basic blocks may have multiple entries in that graph. Also, reachability in the reverse graph is not guaranteed *a priori*. All this implies that the most appropriate treatment of backward analysis will tend to differ considerably from that of forward analysis as described in sections 3 and 4.

In approaching the design of a 'backward' analyzer, a first problem encountered is: where should data-values be computed? An off-hand answer might be: at node exits. Each such exit is associated with an arc (m,n) $\in$ G,where n is a successor of m. (Note that m may contain more than one exit to n, but in our model all these exits are identified, since we do not allow multiple edges between graph nodes.) In such an approach, we would want to compute a data-value $x_{(m,n)} \in L$ for each (m,n) $\in$ G, representing information known upon exit(s) from m to n. Given an edge (n,k) $\in$ G, let $f_{(n,k)} \in F$ denote

the data-flow map representing the effect of reverse flow through n from its exit(s) to k back to its entry. Then we can write a standard set of (intraprocedural) data-flow equations for the values $z_{(m,n)}$ as follows (where x_0 denotes the null data-state that we assume of program exits):

$$
\begin{aligned}
&z_{(m,n)} = x_0 , && \text{if n is a program exit} \\
&z_{(m,n)} = \wedge \{f_{(n,k)}(z_{(n,k)}) : (n,k) \in G\}, && \text{otherwise.}
\end{aligned}
\tag{5.1}
$$

It is easily seen that the maximal fixpoint solution of (5.1) has the property that for each $(m,n) \in G$, $z_{(m,n)}$ is independent of m and depends only on n (since what is known upon exit from m to n is precisely what is known at entry to n). Therefore equations (5.1) are equivalent to the following equations, where for each graph node n, x_n denotes data known at *entry* to n:

$$
\begin{aligned}
&x_n = x_0 , && \text{if n is a program exit,} \\
&x_n = \wedge \{f_{(n,k)}(x_k) : (n,k) \in G\}, && \text{otherwise}
\end{aligned}
\tag{5.2}
$$

In what follows, we will solve equations (5.2) rather than (5.1). Note that by computing data per node rather than per edge we also save a considerable amount of space.

We will now describe an interval-based technique for the solution of equations (5.2) for analyses of the bitvectoring class. This technique is similar in overall design to the technique for forward problems described in Section 3, although significant differences between the two do exist.

As in the forward case, we assume that a preliminary pass through the program code has computed, for each nonvirtual edge $(m,n) \in G$ a data-flow map $f_{(m,n)} \in F$ as defined above. To analyze a single procedure p intraprocedurally, our technique proceeds through the following phases:

(1) Elimination phase

In this phase we iterate through the intervals of p in their reverse preorder (innermost to outermost). For each such interval I we compute two kinds of data-flow maps: (i) For each successor v of I we compute a map $f_{(I,v)}$ describing the effect of flow through I to the entry of v. (ii) For each node $u \in I$ and each successor v of I we compute an *auxiliary map* $\hat{f}_{(u,v)}$ describing the effect of flow from the start of u, through I, to entry to v.

Since equations (5.2) propagate information from successors of a given node back to that node, it follows that in computing the above maps functional composition should be taken in reverse-flow direction. More precisely, the following formulae should be applied in backward analysis:

for auxiliary maps: For each successor v of I, first set $\hat{f}_{(v,v)} = \underline{id}$, and then use the formulae

$$\hat{f}_{(u,v)} = \bigwedge \left\{ f_{(u,w)} \circ \hat{f}_{(w,v)} : (u,w) \in G \text{ and } (w \in I \text{ or } w = v) \right\} \tag{5.3}$$

for each $u \in I$.

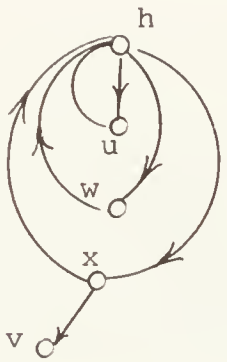
for extended-flow maps:

$$f_{(I,v)} = f_{(I,h)} \circ \hat{f}_{(h,v)} \tag{5.4}$$

for each successor v of I (where h is the head of I).

Equations (5.3) can be solved by iterating through the nodes of I in *reverse* interval order (i.e. in postorder). If I is a proper interval, then *three* iterations through its nodes are required to guarantee convergence of the solution. The reason why an extra iteration (as compared with the forward case) may be required is that, in order to record an event (or anti-event) in $\hat{f}_{(u,v)}$ for some $u \in I$ and v a successor of I, flow has to be traced backward from v to the node w at which this event occurs along an acyclic path,

and then from w to u along another acyclic path. It is easily seen that, for any interval I, this tracing may require up to 2d + 1 reverse iterations through the nodes of I, where d is the loop-connectedness parameter of I, so that for proper intervals (for which d = 1) three iterations might be needed. (As an example consider the following interval I:



In this example, to record an event at node w in $\hat{f}_{(u,v)}$ one has to trace at least the path (u,h,w,h,x,v) which loops back to the head h twice and therefore three iterations are really required.)

In much the same way as in the analysis of the forward case, the above argument implies that for each source of irreducibility imbedded within an improper interval I, two extra iterations through the nodes of I may be required. Even though this doubles the number of extra iterations required to handle imbedded irreducibilities as compared with the forward case, the degradation of algorithm performance will still be very mild.

As to the outermost interval I of p, one reverse iteration through the nodes of I is sufficient if I is proper, and two additional iterations are required per each imbedded source of irreducibility. This interval is treated in a somewhat different manner than the other intervals, as it does not have any successors, so that auxiliary maps cannot be defined for its nodes in the same way as for nodes of inner intervals.

To adjust for this slight difference, we regard the exit block e_p and the stop block s_p as 'successors' of I, which requires us to compute the auxiliary maps $\hat{f}_{(u,e_p)}$, $\hat{f}_{(u,s_p)}$ for each node $u \in I$.

(2) Second elimination phase.

This phase appears only in the solution of backward problems and does not correspond to any phase of the forward algorithms. Our aim in this phase is to compute a second kind of auxiliary map defined as follows: For each node u in the procedure p being analyzed, we compute a map fe_u , describing the effect of reverse flow from the exit(s) of p back to (the start of) u. These maps are essentially the maps ψ_k described in Section 2 for interprocedural purposes, and are analogous to the maps ϕ_n that we compute implicitly in the elimination phase of the forward analysis, but here we prefer to compute them explicitly, since no simple formula analogous to (4.4) exists for direct construction of these maps. (Note, however, that it is only in the interprocedural case that such a computation has to be carried out separately from data propagation. However we separate these two phases even in the intraprocedural case to make our intraprocedural and interprocedural approaches agree. See the next section for more details.)

Computation of the maps fe_u proceeds as follows: We iterate through the intervals of p in their preorder (from outermost to innermost). Consider first the outermost interval I. For each $u \in I$ we can compute

$$fe_u = \hat{f}_{(u,e_p)} \wedge (\hat{f}_{(u,s_p)}(x_0)) \tag{5.5}$$

i.e. take the meet of the map $\hat{f}_{(u,e_p)}$ with the constant map $\hat{f}_{(u,s_p)}(x_0)$. (We treat return and stop blocks differently mainly for interprocedural reasons, since interprocedurally the data-state known at e_p is usually different from the null value x_0 , which is assumed at program *exits*. Note that e_p is not a program exit, whereas s_p is.)

Next suppose that our iteration has come to process an inner interval I. Then for each $u \in I$ we compute

$$fe_u = \wedge \{\hat{f}_{(u,v)} \circ fe_v : v \text{ a successor of } I\} \tag{5.6}$$

Note that any interval containing a successor v of I (i.e. intof(v)) must be an ancestor interval of I, i.e. intof(v) = $intof^k(I)$ for some $k \geq 1$, so that fe_v will already have been computed for all successors v of I when we apply (5.6) to nodes of I.

(A small technical problem arises in connection with *endless loops*. Suppose that L is such a loop, i.e. a strongly-connected program region with no successors. While such loops do not create any problems in forward analysis, since they are reachable from the program (or procedure) entry, they are somewhat problematical in backward analysis, since such loops cannot reach any program exit, and so no information can be propagated to them. Noting that such loops can only appear in the outermost interval, either as a single node (interval) if it is single entry, or as an (irreducible) collection of nodes otherwise, we can solve this technical problem by adding an edge (x,s_p) for each node x in the outermost interval which does not reach e_p or s_p. Assuming this to have been done, the fe maps will then be properly defined at each node of the procedure.)

(3) Propagation phase:

This final phase of backward analysis is relatively trivial in view of the preparation for it accomplished during the second elimination phase. Let $x_p \in L$ be the data-state known at return from a procedure p. (In the intraprocedural case x_p will be the standard null data state x_0 ; in interprocedural analysis, x_p will differ from one procedure to another in the manner described in the next section.) For each node u in p, we compute x_u , the data-state at the start of u, using the simple formula

$$x_u = fe_u(x_p) \tag{5.7}$$

which completes our intraprocedural backward analysis.

6. Interprocedural backward data-flow analysis

In this section we extend the intraprocedural algorithm described in the previous section to the interprocedural case. The required modifications are rather similar to those used for interprocedural forward problems in Section 4, and we will mainly point out the differences between these two techniques. More details are provided in Sections 2 and 4.

Our algorithm consists of the following phases:

(a) Call graph analysis: here we perform exactly the same analysis as is described in Section 4. (As already noted, call graph analysis should be carried out once at the beginning of optimization, prior to any other interprocedural data-flow analysis.)

(b) Initialization: This phase associates data-flow maps $f_{(m,n)}$ of the analysis with each nonvirtual edge $(m,n) \in G$ such that m is not a call block.

(c) Interprocedural elimination phase: In this phase we iterate through the procedures of the program in the same order as is described in Section 4. Each procedure p visited in this iteration is subjected to an elimination pass quite similar to that described in Section 5, which involves adjustment of data-flow maps for call blocks and tests for convergence as in Section 4. Note, however, that in backward analysis the data-flow map ψ_{r_q} describing the effect of flow through a procedure q is computed as

$$\psi_{r_q} = \hat{f}_{(r_q,e_q)} \wedge (\hat{f}_{(r_q,s_q)}(x_0)) \tag{6.1}$$

where x_0 is the assumed null data-state at program exits (compare with (5.5)).

The theoretical bounds on algorithm performance derived in Section 4 apply in this case also. The necessary theorems and proofs are analogous to those given in Section 4, for which

reason we omit them.

(d) Second intraprocedural elimination phase: This is performed for all routines in the program being analyzed, and within each routine uses exactly the technique described in Section 5.

(e) Computation of data at procedure exits: This interprocedural phase is very similar to the phase which calculates entry-data in forward analysis (cf. Section 4). For each procedure p, let $z_p = x_{e_p}$ denote the data state at exit (return) from p. If p is the main program, then (since we assume that the main program is nonrecursive) execution terminates when the exit point of p is reached. Thus we can write the following set of equations, whose heuristic meaning should be self-explanatory:

$$\begin{aligned} z_{main} &= x_0 \\ z_p &= \wedge \{fe_v(z_q): \text{ v follows immediately a call to p in q}\} \end{aligned} \tag{6.2}$$

As in Section 4, it is convenient to transform (6.2) into a data flow problem for the call-graph, by defining, for each $(p,q) \in CG$

$$g_{(p,q)} = \wedge \{fe_v: \text{ v follows immediately a call to q in p}\}. \tag{6.3}$$

This transforms (6.2) into

$$\begin{aligned} z_{main} &= x_0 \\ z_p &= \wedge \{g_{(q,p)}(z_q): (q,p) \in CG\} \end{aligned} \tag{6.4}$$

Note that equations (6.4) define a *forward* data-flow problem rather than a backward problem for CG. This can be explained as follows: In the forward case, information is propagated from the entry of a calling procedure to the entry of a called

procedure. In the backward case, information is propagated backward from the exit of a calling procedure to the exit of a called procedure; but both propagations induce a forward propagation in the call graph.

The maximal fixpoint solution of (6.4) is found precisely as in Section 4, using the same iteration order over procedures, so that the performance bounds described in Section 4 remain valid.

(f) Final propagation phase: This phase is very similar to the final intraprocedural propagation phase described in Section 5. For each procedure p, let z_p be the data state at its return point, as computed in phase (e). Then for each node u in p we compute the data-state x_u at the start of u as follows:

(6.5) $$x_u = fe_u(z_p)$$

This completes our analysis.

7. Code motion

In this section we describe a technique for performing code motion as part of a data-flow analysis. The technique is surprisingly simple and straightforward, but it rests on heuristic assumptions concerning the safety and profitability of motion which are quite adequate for SETL, but may not be generally applicable.

That code motion and data-flow analysis are closely related can be seen by considering the typical case of available expressions analysis. In this analysis a 'T-event' is a computation of an expression T, and a corresponding 'anti-event' is a re-definition of a variable on which T depends. In this analysis we determine for each program point n the expressions T which have the property that every path leading from program entry (or procedure entry) to n contains an event for T which is not followed by an anti-event. If T is such an expression, then a computation of T at n is *redundant* and can be eliminated. In more general cases this recomputation may not be unconditionally redundant, but may become so if we insert another earlier event for T at a point having lower execution frequency. When this is the case it may be profitable to insert a preceding event (i.e. computation), thereby eliminating the need to compute T at n and reducing the total number of computations of T. This code transformation is known as *code motion* even though, strictly speaking, nothing is really moved. (See [AU], [Sc], [MR], [MFS] for various code motion algorithms.)

Abstracting from this example, we say that a (forward) data-flow analysis of the bitvectoring class is *amenable to code motion* if it has the property that whenever an event occurs at a program point n and any path from the program entry to n contains a similar event not followed by a corresponding anti-event, then the event at n is redundant and the actual operation(s) which realize this event can be eliminated without changing the overall program behavior. Among the data-flow analyses amenable to code motion we may mention:

(a) Copy optimization, in which an event occurs whenever we create an unshared value for an object (either by copying the object or by computing a new value for it) and where any operation causing the value of the object to be shared is viewed as the corresponding anti-event. (Note, however, that in this case, only value copying events can be redundant.)
(b) Conversion optimization. This analysis appears (e.g. in the SETL optimizer) when program objects can be given more than one data-representation, making it necessary to determine where conversions (or checks) between different representations must be inserted. In this analysis an event (for a specific variable V and a specific representation R) corresponds to the conversion of V to the representation R or to the assignment of a value having that representation to V; and each conversion of V to any representation other than R as well as each assignment to V of a value having representation different from R counts as an anti-event for V and R (see Section 9 for more details).

Any code motion (or, more correctly, code insertion) involves the two preconditions of *profitability* and *safety*. It is profitable to insert code at some program point if this insertion will make other code redundant and decrease expected (or, using a stricter criterion, absolute) execution time. It is safe to insert code at a program point if insertion cannot cause program abort except in cases where the original program would have also aborted. We refer the reader to [AU], [Sc] or [MFS] for a discussion of various possible criteria ensuring safety and implying profitability. In what follows we will simplify our presentation by ignoring the issue of safety, and simply assume that any code insertion which we may want to make is safe. (In SETL this assumption can be guaranteed by executing the optimized program in a special run-time error mode.) However, we note that the following

discussion generalizes easily to cases in which safety must be enforced, though in such cases it will usually be necessary either to perform an additional *backward* data-flow analysis to assess the safety of code insertion, or to restrict code insertion to cases in which safety is assured a priori, e.g. only allow code insertion at entries to loops (intervals) which must be executed at least once, and which are such that every path through such a loop contains the calculation which we wish to insert at loop entry.

As to profitability, we will attempt to insert code only at interval preheaders, and will assume that it is profitable to insert code at entry to an interval I if there exists at least one corresponding event in I which this insertion makes redundant. Of course, such code insertion can increase execution time rather than decrease it in certain unlikely cases, e.g. if the eliminated calculation is bypassed when I is executed. However, if we assume that, on the average, all code within I is executed at least once whenever I is entered, then our profitability criterion is seen to be quite reasonable.

To facilitate code motion we shall compute one additional data object, a map z. Initially, z maps each basic block n to the set z_n of all T such that n contains a T-event which becomes redundant if the same T-event is available at entry to n but not otherwise. (For basic blocks these are precisely the T for which there exists an *upward-exposed* T-event in n, i.e. a T-event not preceded by any T-anti-event.) By extending the map z to intervals we will be able to determine the code which should be inserted at each interval entry. (Note that since intervals are identified with their own preheaders, the map values z_I, where I is a (preheader of an) interval, are also known initially, but only reflect flow through the preheader itself.)

Consider the elimination phase of the forward data-flow analysis algorithm described in Section 3. Let E be the underlying universal set for the analysis, i.e. the set of all elements over which bit-vectors are taken. Let I be an innermost interval (i.e. an interval not containing any subinterval). Then, for each node $n \in I$ we reason as follows (where the auxiliary maps $\hat{f}_n$ reflect only flow through the loop of I but not through its prehader; see Section 3):

(a) $\hat{f}_n(E)$ is the set of all elements T such that if a T-event is available at entry to the loop of I it remains available at entry to n.

(b) $\hat{f}_n(\emptyset)$ is the set of all elements T such that the T-event is unconditionally available at entry to n even if it is unavailable at entry to the loop of I.

(c) Hence, $\hat{f}_n(E) - \hat{f}_n(\emptyset)$ is the set of all T whose events are available at entry to n if and only if they are available at entry to the loop of I.

(d) Therefore, $[\hat{f}_n(E) - \hat{f}_n(\emptyset)] \wedge z_n$ is the set of all T such that n contains a T-event which becomes redundant if and only if such an event is made available at entry to the loop of I.

(e) This implies that if we define

$$\text{insert}(I) = \bigvee \{[\hat{f}_n(E) - \hat{f}_n(\emptyset)] \wedge z_n : n \in I\}$$

(where $\bigvee$ denotes set union).
Then insert(I) is the set of all T such that I contains a T-event which becomes redundant if T is made available at entry to the loop of I but not otherwise. According to our profitability criterion, this is the set of all computations to be inserted at entry to the loop of I (i.e. at the end of the preheader of I).

In order to be able to repeat these arguments for intervals I containing and contained in other intervals, we must define the map values z_I appropriately. For each interval I we wish z_I to be the set of all T such that I contains a T-event which becomes

redundant if a T-event is inserted at entry to I but not otherwise. (These are very similar to the weakly potentially redundant expressions discussed in [MFS].) By an argument similar to (d) above it can be easily seen that this is accomplished by modifying the value z_I attached to the preheader of I as follows:

$$z_I \leftarrow z_I \vee \{[f_{(I,h)}(E) - f_{(I,h)}(\emptyset)] \wedge \text{insert}(I)\}$$

(where h is the interval head, and as usual I is identified with its own preheader). If the preheader of I is empty, this last expression reduces to insert(I).

By carrying this process iteratively for all intervals from innermost to outermost, we can compute insert(I) for all intervals I (with the exception of the outermost interval J, for which code motion is pointless since J does not represent a loop). The necessary processing can be incorporated either in the elimination phase of the data-flow algorithm, or in a separate pass just after the elimination phase (see also a remark below). A similar approach to code motion has been devised in [MFS].

The value insert(I) defines the code to be inserted at entry to the loop of I. However, some of this code may be redundant (either unconditionally, or because of code motion out of some interval containing I) at its nominal point of insertion. With this in mind, we delay actual code insertion (motion) till the propagation phase of our algorithm. In this phase we iterate over intervals from outermost to innermost. When processing an interval I during this phase we will already have computed the attribute data x_I known at its entry. This allows us to compute

$$\hat{x}_I = f_{(I,h)}(x_I) \quad \text{(h is the head of I)}$$

to obtain data known at the end of the preheader of I (i.e. the program point at which code moved out of I ought to be inserted. We can then compute

(a) insert(I) ← insert(I) - $\hat{x}_I$

(b) $\hat{x}_I \leftarrow \hat{x}_I$ + insert(I)

This gives us i) a modified descriptor insert(I) defining the code that actually has to be inserted into the preheader of I, and ii) modified attribute data $\hat{x}_I$ at entry to the loop of I, which takes the newly inserted code into account. (Note that we assume here that the only effect of the newly created code on our analysis is to make computations available and that no other elements (bits) are affected by it.) After computing $\hat{x}_I$ we then use it to propagate attribute data to nodes of I.

This completes the description of our code motion algorithm. However, several related issues still deserve comment. Note first that if I is an interval, then availability of the computations inserted at the preheader of I is exploited only in propagating data to nodes of I, but not to update any flow-maps describing flow through intervals containing I. Suppose, for example, that we process the following code:

```
(1)  (while ...)
(2)       (while ...)
(3)          a * b
(4)       end while;
(5)       a * b
(6)   end while;
```

Then although we move the computation of a * b at line (3) out of the inner loop and thereby make the computation at line (5) redundant, our algorithm will fail to detect this fact.

Our reason for not using the 'insert' map in this more extensive manner suggested by this example is that the expression insert(I), is *not monotone* in the functions $\hat{f}$. Thus if I is processed repeatedly, as will be the case if I lies on a recursive cycle in an *interprocedural* analysis, then use of the insert(I) (or, more precisely, the z_I) information to update flow-maps might cause the algorithm to diverge. However, in

intraprocedural analysis, and even in interprocedural analysis of recursion-free programs, in which each interval is processed precisely once in the elimination phase, we could improve the results of the analysis by incorporating the 'insert' information into the flow maps. This can be done simply as follows: Let I be an interval with head h. Once I has been processed, we could put

$$f_{ins} \leftarrow [E, \text{insert}(I)]$$

$$f_{(I,h)} \leftarrow f_{ins} \circ f_{(I,h)}$$

and then use this modified $f_{(I,h)}$ to compute the flow maps for all virtual edges going out of I. However, this improvement is probably only marginal.

A related problem is that of *interprocedural code motion*. Here new and interesting possibilities which deserve further study arise. In particular, the techniques we have described can be used to move code out of a routine. Consider, for example, the following code:

```
(while ...)          proc p;
   call p            a * b
end while;           end proc;
```

Here, if we insert a * b at entry to the while loop (and provided of course that a and b are globals, or are made into globals), then a * b becomes redundant in p. Interprocedural motion can be accomplished while processing the calling routine (and after p has been processed), by assigning z_{r_p} to z_c, where c is the call block containing the call to p, and r_p is the entry to the procedure p (identified with the outermost interval of p). This will make a * b 'upward-exposed' in c, from which it can then be further moved to the preheader of the while loop. However, if p is also called from other points at which motion of a * b is not feasible, then insertion of a * b at

entry to the while loop will not make a * b redundant in p. Two approaches to this situation are possible: (i) We can make call blocks targets for code motion. That is, if c is a call block, then we can require all computations in z_c to be inserted at entry to c, unless already available there. This approach is capable of moving code out of a procedure to all call points to the procedure, and even of moving some of this code further away. (ii) We can attempt interprocedural code motion only in situations where it is possible to test for availability of a computation at run time and skip recomputation if it is already available. This is the case for the SETL *copy optimization*, where dynamic test of a 1-bit reference count is possible. In this case motion of a copy operation out of p to the entry to the while loop shown in the above example can eliminate the need to copy inside p when p is called from within that while loop, even though copying within p may still be required if p is called from other points.

Either of these approaches must be used with caution for recursive procedures, since in a recursive cycle of calls it becomes much more difficult to assess the profitability of code motion between procedures.

8. Bit-Matrix Data Flow Problems

The 'bitvectoring' analyses that we have considered in the preceeding pages are the simplest of those which belong to the general class of flow analyses introduced by Kildall [Ki]. Their defining property is that they deal with boolean attributes that do not interact; i.e., each program statement that effects an attribute either sets it or drops it. This special property is essential to the very efficient interprocedural analytic techniques which we have outlined. However, interest exists in many other, less entirely trivial analyses,and it is useful to review the simplest of these and comment on the efficiency with which they can be carried out.

An important class of optimizations lying just beyond the elementary bitvectoring class is that in which variable attributes are still boolean, but in which the effect of code can either be to set, drop, or transfer an attribute. As an example, consider a hypothetical language in which assignments can transfer pointers, and suppose that we wish to determine all variable occurences at which a given pointer or a member of some well-defined class of pointers can appear. In this situation it is natural to work with the attribute 'can-be-pointer'. Some assignments (e.g. of non-pointer constants) will clearly kill this attribute, while others (e.g., of explicit pointers) will set the attribute; but the interesting new fact is that various assignment operators, including simple assignments

```
a := b
```

will transfer the 'can-be-pointer' attribute from b to a. To take this into account the effect of program flow must be described, not by a pair of bitvector coefficients as before, but by a linear boolean mapping $f(x) = Ax + b$. Here A is not a bitvector of

length n (where n is the number of variable occurences entering into the analysis), but an n x n boolean matrix. This makes map composition much slower than the bitvector operations on which we could rely in the preceding sections. Moreover, an elimination approach like that which we have described becomes infeasible, because of the large amount of data that would have to be stored to keep coefficient matrices A available at many program points. Thus, if we admit even this minimal complication of the situation in which a bitvectoring approach is possible, analysis immediately becomes much more difficult, even though the form of the equations defining the analysis changes very little.

It is also worth noting that many more iterations may become necessary to attain convergence in this case than are necessary in the bitvectoring case. For example, consider the following code:

```
label:   x1 := x2;
         x2 := x3;
         . . .
         xn-2 := xn-1;
         xn-1 := xn;
         xn := pointer;
         go to label;
```

This loop must be iterated n times for the pointer value assigned to x_n to propagate to x_1. The situation that confronts us here resembles the problem of forming the transitive closure of a boolean vector under a general boolean matrix. Of course, even in this case we can generally expect to propagate a single boolean attribute to all relevant parts of a program in time roughly proportional to program length by proceeding along chains of 'nearest occurrences' of a single variable. However,

each attribute would have to be traced separately since no effective algorithm allowing parallel analysis of a whole group of attributes is known in this case. It should be noted that these inefficiencies result from the dependence between different attributes that we have assumed. If independence of attribute propagation in an analysis is assumed, even non-boolean attributes could be analyzed by elimination techniques like those described in earlier sections, with only relatively mild degradation of performance (see [RO_1] for example).

Overall, we come to the pessimistic conclusion that to carry out program analysis effectively by presently known algorithmic techniques it is necessary either to confine oneself to analyses which can be forced into a bitvectoring mold (or, more generally, which deal with independent simple attributes); to analyse for relatively small numbers of more interdependent boolean attributes; or to work with attributes for which a crude iterative technique converges more rapidly than worst-case theoretical arguments would lead one to expect.

9. Applications of the General Algorithms in The SETL Optimizer

In this section we will describe the specific bit-vector data-flow problems arising in SETL optimization which are solved using the general-purpose package of algorithms described in sections 3-7.

I. Available Expressions Analysis. This well-known analysis is performed as follows. With each well-defined expression e having no side effects we associate a variable v_e which is used to store the value of e whenever e is computed. A redundant computation of e can then be characterized by the property that the value of v_e at a point of computation of e is always equal to the result of computing e, so that instead of computing e we can simply fetch and use the value of v_e. (The value of a non-redundant computation of e may then have to be stored in v_e if this value will be used at some subsequent redundant computation of e. We can determine whether such a store is necessary either by a live-dead analysis, or more simply by using a modified use-definition chaining map (see below).)

Available expressions analysis is performed as follows. As an analysis framework we use the lattice $L = 2^E$, where E is the set of all well-defined expressions having no side effects. Meet in L is taken to be set intersection. Each $x \in L$ denotes a set of expressions available at some program point n, i.e. expressions e having the property that along every execution path leading from the program (or procedure) entry to n, e has been computed ('generated') with no subsequent modification of the variables on which e depends (i.e. no 'kill' of e). The set F of data-propagation maps of the analysis consists of functions $f : L \to L$ having the form

$$f(x) = (thru_f \cap x) \cup gen_f, \qquad x \in L$$

where $thru_f \in L$ is the set of all expressions e, which, if available at the start of the flow described by f, are also available at the end of that flow, and where $gen_f \in L$ is the set of all expressions which are unconditionally available at the end of that flow.

We invoke the algorithms described in sections 3,4 and 7 to perform redundancy analysis (which is a forward analysis) and code motion. Note that the interprocedural part of the analysis only needs to deal with expressions which depend on at least one global variable; all 'strictly local' expressions are analyzed separately, each within its own procedure.

For code motion we apply the algorithm of section 7 as it stands, ignoring the issue of safety altogether. This is possible since SETL will execute programs in a special run-time error mode for which erroneous computations do not cause program abort, but rather yield a special 'error' value. Once generated, error values will propagate through other computations as long as they are not used in branch instructions, in which case the program does abort. It is easily seen that this treatment of errors allows us to insert computations safely at any program point.

The solution map x generated by our algorithms defines the set of expressions available at entry to each basic block n. An additional scan through all blocks will then detect redundant computations and eliminate them, and also insert movable code into interval preheaders.

II. Modified Use-definition Chaining Calculation. In this analysis, which prepares data-structures used in later optimizer phases, we compute a variant of the well-known use-definition map (cf. [A1]), which we denote as 'bfrom', and which is defined so that for each use vo of a variable V, bfrom{vo} is the set of all other occurrences (definitions and uses) of V

from which vo can be reached along a path clear of all other occurrences of V. The classical use-definition map is actually the transitive closure of bfrom, but we use bfrom instead since we expect this to speed-up subsequent attribute-flow analyses (mainly type analysis), and because the bfrom map is more suitable than the use-definition map for various other optimizations (such as dead-code elimination).

To calculate the bfrom map, we perform a reaching occurrences analysis. In this analysis, for each basic block n we compute the set x_n of all variable occurrences vo which can reach the start of n, i.e. for which there exists a path leading from vo to the start of n which is clear of any other occurrence of the same variable. This is a forward analysis which uses the semilattice $L = 2^E$, where E is the set of all occurrences of relevant program variables. (Again global variables have to be analyzed interprocedurally, whereas local variables are analyzed intraprocedurally, each within its own routine.) The meet in L is set union.

The space F of data-propagation maps consists of functions $f: L \to L$ having the form

$$f(x) = (thru_f \cap x) \cup reachin_f, \qquad x \in L$$

where $thru_f \in L$ is the set of all variable occurrences vo for which there exists a path through the flow described by f which is either free of any occurrences of the associated variable V, or else contains vo as the last occurrence of V, and where $reachin_f \in L$ is the set of all variable occurrences vo for which there exists a path through the flow of f which contains vo as the last occurrence of V.

After analysis is carried out using the algorithms from section 3 and 4 (code motion is obviously meaningless for this analysis) the computation of bfrom is completed by a straightforward scan through all basic blocks.

III. Copy Optimization. SETL is a value language, but for efficiency its value semantics is implemented using pointers. This usually requires values to be copied before being modified if they are shared by (i.e. pointed to by) several variables. In our implementation of SETL, a certain part of excess value copying is suppressed by 1-bit reference counts, known as 'share-bits', attached to each variable. The share bit of a value is set whenever a value is shared (which can happen in consequence of an assignment, imbedding or retrieval operation), and is dropped whenever a variable is assigned a newly created value. This mechanism, though crude, does suppress most redundant copy operations at run-time. To improve program performance still further, the SETL optimizer includes a copy optimization phase whose goals are as follows: (a) To detect potentially destructive value uses at which copies will never be required and eliminate the dynamic testing of the share bit; (b) To detect cases at which a copy will always be required at a use, suppress share bit testing, and emit an unconditional copy instruction just before that use, (c) To suppress setting of share-bits that are never going to be tested (either because there occur no subsequent destructive uses of a particular value, or because subsequent dynamic tests of a share-bit have been eliminated by (a) and (b) above); (d) To move copy instructions out of loops.

To achieve these goals we proceed as follows. First we perform available unshared values analysis. In this forward analysis we compute, for each basic block n, the set 'unshared(n)' of all variables whose value is definitely unshared at entry to n; in addition we use the code motion algorithm to move copy operations out of loops. The framework for this analysis involves a lattice $L = 2^E$, where E is the set of all relevant program variables and where lattice meet in L is set intersection; also a space F of data-propagation maps, where each $f \in F$ has the form

$$f(x) = (thru_f \cap x) \cup newin_f, \qquad x \in L.$$

Here $thru_f$ is the set of all $V \in E$ such that each path through the flow of f is either free of any set/drop of the share-bit of V or contains a drop of that share-bit not followed by any setting of it, and $newin_f$ is the set of all $V \in E$ such that each path through the flow of f contains a drop of the share-bit of V not followed by any setting of that bit.

In addition, to facilitate code motion, for each basic block n we compute the set exposed(n) of all $V \in E$ such that n contains a potentially destructive use of V not preceded by a set or drop of the share-bit of V.

This analysis is performed using the algorithms of sections 3, 4 and 7, and allows us to carry out the optimizations (a) and (d) mentioned above.

To accomplish goal (b) we perform a dual forward analysis, called available shared values analysis, in which for each basic block n we compute the set 'shared(n)' of all relevant variables whose value is definitely shared at entry to n. This analysis is performed in exactly the same way as the preceding analysis (but without code motion), simply by reversing the roles of share-bit drops and settings. For each remaining copy operation C this analysis determines whether C is conditional (involving share-bit testing), or unconditional, (i.e. whether the value copied by C is definitely shared); if C is unconditional, dynamic share-bit testing is suppressed.

Finally, to accomplish goal (c), we perform a backward analysis which, for each program point, computes the set of all variables V whose share-bit value at that point reaches a point where it is tested along a path free of any operation which sets/drops that bit. For each share-bit setting S this analysis determines whether S is really required, and suppresses S if the bit S sets is not going to be tested subse-

quently. This last analysis can be viewed as a special case of live-variables analysis of the kind described in part V of this section, and is performed using essentially the same method as outlined there.

IV. Conversion Optimization. This analysis is required as a final step in our type-analysis and data-representation selection phases. Since SETL is dynamically typed, variable values can acquire more than one data-type or representation during program execution. Consider for example the case in which during execution a variable V acquires values having data-representation R_1 and also values having the representation R_2. Without optimization, this will require the compiler to treat V as having a rather general data representation, and consequently to emit somewhat inefficient code, both because (i) instructions manipulating V will have to be less specific (e.g. off-line general addition vs. the much more efficient in-line integer addition), and because (ii) data-type checks and conversions may be required, prior to instructions manipulating V.

The optimizer can eliminate some of these inefficiencies by associating a data-representation with each variable occurrence in the program being analyzed, and then splitting each variable V into a series of variables V_{R_1}, V_{R_2},..., having representations R_1, R_2,..., respectively, where R_1, R_2,..., are more specific representations computed for the occurrences of V, and where all these variables share a common 'cell' in storage. Then each occurrence of V having computed representation R can be replaced by an occurrence of the 'split-variable' V_R. This technique enables generation of more specific, and therefore more efficient instructions to manipulate V. However, it will also give rise to situations in which two different variables V_{R_1}, V_{R_2} split from the same variable V are linked in data flow (e.g. V_{R_1} is defined and V_{R_2} is then used). In such cases

we must make sure that the value of V at the second use does indeed have the required representation R_2. If unable to guarantee this assertion at compile time, we must insert an explicit data-type check/conversion of V to R_2 preceding the second use. This is accomplished by our conversion optimization phase.

To accomplish this task we perform a bitvectoring data-flow analysis called 'available conversions' analysis which, for each basic block n determines the set x_n of all split variables V_R which are 'available' at the start of n, i.e. whether each execution path leading to the start of n contains an occurrence of V_R which is not followed by any other occurrence of V. In conjunction with this analysis we use the code motion algorithm to move data-type checks and conversions out of loops.

This analysis uses the following framework. The lattice L used is 2^E, where E is the set of all relevant split-variables, and where meet in L is set intersection. (We make the same separation between global and local variables as in reaching occurrences analysis.) The space F of data-flow maps used consists of functions $f: L \to L$ having the form

$$f(x) = (x \cap thru_f) \cup gen_f, \qquad x \in L$$

where $thru_f \in L$ is the set of all split-variables V_R which, if available at the start of the flow described by f, will also be available at the end of that flow. That is, each path through the flow of f must either be free of any occurrences of V, or else the last occurrence of V along that path must have the representation R (i.e. must be an occurrence of V_R).

Moreover, $gen_f \in L$ is the set of all split-variables V_R which are unconditionally available at the end of the flow described by f, i.e. $V_R \in gen_f$ if each path through the flow corresponding to f contains an occurrence of V_R not followed by any other occurrence of v.

In applying our 'forward' algorithms to this framework, we have to consider the safety of conversion motion (insertion). Unlike insertion of ordinary computations, insertion of a conversion to a representation R can be unsafe (i.e. may cause a new program abort). As an example, consider the following code.

```
read (V);
(while . . .)
     if C then
          V := V + [x];
     end if;
end while;
print (V);
```

Here, the code motion algorithm of section 7 would suggest moving V_{tuple} out of the while loop (i.e. would insert a conversion of V to tuple form at the loop preheader). However, as read V might be an integer, and the condition C might be a type test to skip the concatenation operation in this case. With these suppositions the original program would not have aborted, but the modified program will abort.

It is therefore necessary to perform a preliminary safety analysis before applying the code motion algorithm. This is a backward-union bitvectoring analysis, which for the start of each basic block n, determines the set y_n of all split-variables V_R which can safely occur at that point, i.e. calculates those V_R for which all paths forward from the start of n onward either lead to a use of some V_{R_1} (with no intervening occurrences of V), where R_1 is either equivalent or more general than the representation R (so that conversion of V_R to V_{R_1} will always succeed), or else leads to a program exit, or to a re-definition of V (with no intervening occurrences of V). The framework for this safety analysis is constructed similarly to the framework of the available conversions analysis.

Having determined the sets y_n, we can handle safety of code motion as follows: Let V_R be a split-variable that we wish to insert at (the end of) a preheader of a loop having entry node n. Then it is safe to insert V_R at that point if every variable V_{R_1} split from V and belonging to y_n has a representation which is either equivalent to or more specific than R.

Note also that, in conversion motion as in expression motion, the changes resulting from the insertion of conversions at an interval's preheader I need not be propagated globally to flows in intervals containing I. Such propagation is required only if it could be necessary to perform a conversion that would have been unnecessary in the original program, or if propagation would prevent additional conversion motion that would have been possible in the original program. However, it follows from the special nature of our code motion algorithm that these cases cannot occur. The proof is not difficult, but somewhat lengthy and technical, and is omitted.

Conversion optimization uses a linear scan of the code in which we compute the sets availconv(I) of all split-variables available just before an instruction I. If I uses some split-variable V_R which does not belong to availconv(I), a run-time check or conversion into the V_R form is required before I, and is inserted there; otherwise no such conversion is required. Additionally this third step inserts conversions at loop preheaders in a manner controlled by the result of the code motion phase.

Note that our algorithm does not allow us to produce the most specific diagnostic messages when an error occurs. Indeed, forward analysis only tells us whether or not a conversion is required at a given point P, but does not calculate the data representations possible for a variable at P. To gather this additional information, our forward analysis would have to be replaced or augmented by a forward-union analysis, in which we calculated all possible split variables V_R which can reach a particular program point. Such analysis(quite similar to reaching occurrences analysis) would provide this extra information. In the SETL optimizer such diagnostic messages are not required at this step, as they are produced during the type-analysis phase. However, if one wished to adapt the techniques that we have sketched to other kinds of code motion and elimination (e.g. elimination and motion of range checks), such extra analysis might be appropriate.

V. Live-Dead Analysis

This classical analysis ([He],[AU]) establishes the live/dead status of variables. A variable V is said to be live at a program point n if there exists a path leading from n to some use of V which is free of any other occurrence of V (implying that the current value of V may be used subsequently, and so cannot be destroyed or discarded); otherwise V is said to be dead at n.

Live-dead analysis has many well-known applications, such as (a) Register allocation during code generation, since only live variables need be put in registers, (b) Dead code elimination, since operations whose output is dead upon their completion can be eliminated, (c) Static storage allocation optimization, and (d) Various peephole optimizations such as replacement of the sequence 'T := exp; A := T;' by 'A := exp;' provided that T is dead at the end of that sequence.

Live variable calculation is a backward-union analysis, and is performed straightforwardly using the algorithms of sections 5 and 6. The framework used involves the lattice $L = 2^E$, where E is the set of all relevant program variables, and where lattice meet is set-union. Each propagation map f acting on L has the form

$$f(x) = (thru_f \cap x) \cup livein_f, \qquad x \in L$$

where '$thru_f$' is the set of all variables $V \in E$ for which there exists a path through the flow of f which is either free of any occurrence of V or else contains a use of V not preceded by any other occurrence of V, and where '$livein_f$' is the set of all $V \in E$ for which there exists a path through the flow of f which contains a use of V not preceded by any other occurrence of V.

The output of live-dead analysis is a map 'liveat', mapping each basic block n to a set liveat(n) of all variables live at the start of n. These sets can then be propagated (backward) through basic blocks to establish variable liveness at any required program point.

REFERENCES

[AHU] Aho, A. V., Hopcroft, J. E. and Ullman, J.D., "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1974.

[AU] Aho, A. V. and Ullman, J. D. "Principles of Compiler Design", Addison-Wesley, 1977.

[Al_1] Allen, F. E., "Control-flow Analysis", Proc. Symp. Compiler Optimization, SIGPLAN Notices 5 (1970), 1-19.

[Al_2] Allen, F. E., "Interprocedural Data-flow Analysis", Proc. IFIP (1974), North-Holland, 398-402.

[AC] Allen, F. E. and Cocke, J., "A Program Data-flow Analysis Procedure", CACM 19 (1976), 137-147.

[Ba] Banning, J. P., "An Efficient Way to Find The Side Effects of Procedure Calls and The Aliases of Variables", Proc. 6th POPL Conference (1979), 29-41.

[He] Hecht, M. S., "Flow Analysis of Computer Programs", Elsevier North-Holland, New York, 1977.

[HU] Hecht, M. S. and Ullman, J. D., "A Simple Algorithm for Global Data-flow Analysis Problems", SIAM J. Computing 4 (1975), 519-532.

[KU] Kam, J. B. and Ullman, J. D., "Global Data-flow Analysis and Iterative Algorithms", JACM 23 (1976), 158-171.

[Ke] Kennedy, K., "Node Listings Applied to Data-flow Analysis", Proc. 2nd POPL Conference (1975), 10-21.

[Ma] Markstein, P., Private Communication, 1978.

[Ro_1] Rosen, B. K., "Monoids for Rapid Data-flow Analysis", IBM Research Report RC-7032, Yorktown Heights, 1978.

[Ro_2] Rosen, B. K., "Data-flow Analysis for Procedural Languages", to appear in JACM, 1979.

[Sc] Schwartz, J. T., "On Programming: An Interim Report on the SETL Project", 2nd Edition, Courant Institute of Math. Sciences, New York, 1975.

[SS] Schwartz, J. T. and Sharir, M., "Tarjan's Fast Interval-finding Algorithm", SETL Newsletter # 204, Courant Institute of Math. Sciences, 1978.

[SP] Sharir, M. and Pnueli, A., "Two Approaches to Inter-procedural Data-flow Analysis", to appear in "Program Flow Analysis: Theory and Applications", Prentice-Hall.

[Ta] Tarjan, R. E., "Testing Flow-graph Reducibility", J. Comp. Sys. Sciences 9 (1974) 355-365.

[MFS] Mintz, R., Fisher, G. A. and Sharir, M., "The Design of a Global Optimizer," Proc. SIGPLAN Symposium on Compiler Construction, August 1979.

[MR] Morel, E. and Renvoise, C., "Global Optimization by Suppression of Partial Redundancies," CACM 22 (1979) 96-103.

APPENDIX A : SETL Code for the Data-Flow Algorithms

This appendix contains the SETL code for the various control flow and data flow algorithms described in this paper, as taken from the SETL optimizer. The code is divided into two packages (modules) - a control-flow analysis package containing the interval analysis algorithm and related routines, and a data-flow analysis package containing the various data-flow analysis algorithms described in this paper.

The code has been tested on a variety of test programs. The data-flow package has been used for available expression analysis, reaching occurrences analysis and live-dead analysis. (Note however that the code below does not contain the preparatory phase of data-flow analysis in which the initial set of the analysis data-flow maps is computed, nor does it show either the actual invocation of the general data-flow algorithms described in this paper or the concluding phase which utilizes the results of the analysis performed. All these phases vary substantially from one analysis to another, and are therefore left out.)

The code given here is slightly modified from the original optimizer code. The modifications are generally character-set changes, documentation upgrades and omissions of certain code segments which deal with details particular to the SETL intermediate code representation.

```
    MODULE SETL_OPTIMIZER - INTERVAL_ANALYSIS;

$ This module contains the interval analysis algorithm described
$ in SECTION 3.

$ Flow graph analysis produces two maps which serve as input to the
$ interval analysis:

$ 1. CESSOR:      The successor map for basic blocks

$ 2. PRED:        The predecessor map for basic blocks

$ Interval analysis produces five maps:

$ 1. INTOF:       A map from each node to the interval immediately
$                 containing it.

$ 2. INTS:        Maps each routine to a tuple of all its intervals
$                 in reverse preorder. Note that iterating over
$                 INTS(ROUT) is equivalent to iterating from innermost
$                 to outermost interval.

$ 3. INT_NODES:   A map sending each interval into a tuple containing
$                 the nodes of the interval in reverse postorder.
$                 Iterating over INT_NODES(I) is equivalent to iterating
$                 forward over the nodes in I.

$ 4. PROPER_INTS: The set of proper (reducible) intervals.

$ 5. VEDGES:      The set of all virtual edges added to the flow graph
$                 during interval analysis. A virtual edge is an edge
$                 having the form (I, V), where I is an interval
$                 and V is a node outside I which is a successor of
$                 some node in I.

$ All these variables are assumed to be globally accessible in the
$ SETL optimizer.  Additional global variables that are accessed in
$ this module are:

$ ROUTS:        Set of all routines in the program being analyzed.

$ RENTRY:       Maps each routine to its entry block.

$ REXIT:        Maps each routine to its exit (return) block.

$ RSTOP:        Maps each routine to its stop block, if it exists.

$ ROUTOF:       Maps each basic block to the routine containing it.
```

```
$ The module contains three principal routines:

$ 1. FIND_INTERVALS:  Iterates over 'ROUTS' calling other routines

$ 2. GET_GRAPH:       Builds a flow graph for a routine.  Code for this
$                     routine is omitted, since it is largely trivial
$                     and contains many details special to the SETL
$                     language.

$ 3. FIND_INTS:       Finds the intervals of a flow graph.

$ The following variables are used globally during interval analysis:

     VAR
          NODENO,       $ Preorder node numbering
          POSTNO,       $ Postorder numbering
          NDESCS,       $ Number of descendants of each node
          NODES,        $ Tuple of nodes in preorder
          POSTNODES,    $ Tuple of nodes in postorder
          NPRE,         $ Current position in preorder numbering
          NPOST,        $ Current position in postorder numbering
          SEEN,         $ Nodes already in spanning tree
          IMPROPERS;    $ Set of 'heads' of multiple entry loops
```

```
    PROC FIND_INTERVALS;

$ This routine iterates over all the routines in a SETL program
$ finding the interval graph for each routine.

$ Initialize all output objects
    INTOF := INTS := VEDGES := PROPER_INTS := INT_NODES := {};
    CESSOR := PRED := {};

    (FORALL R IN ROUTS)
        GET_GRAPH(R);
        FIND_INTS(R);
    END FORALL;

    PRINT(' ');
    PRINT('   I N T E R V A L   A N A L Y S I S');
    PRINT(' ');

    PRINT('INTS =', INTS);
    PRINT('INT_NODES =', INT_NODES);
    PRINT('PROPER_INTS =', PROPER_INTS);
    PRINT('VEDGES =', VEDGES);
    PRINT('INTOF =', INTOF);
    PRINT('CESSOR =',CESSOR);
    PRINT('PRED =',PRED);

    END PROC FIND_INTERVALS;
```

```
    PROC FIND_INTS(R);

$ This routine calculates the intervals of an intraprocedural flow
$ graph corresponding to a given routine R.

$ FIND_INTS is called once to process each procedure 'R'.
$ it produces five maps:


$ 1. INTOF:       A map from each node to its interval.

$ 2. INTS:        A map sending each routine 'R' into a tuple
$                 containing the intervals of 'R' in reverse preorder.
$                 Note that iterating backward (forward) through
$                 INTS(R) is equivalent to iterating from outermost
$                 to innermost(innermost to outermost) interval.

$                 The outermost interval is not really an interval
$                 at all. Instead it contains all nodes not contained
$                 in other intervals. It is acyclic in the reducible
$                 case.

$ 3. INT_NODES:   A map sending each interval into a tuple containing
$                 the nodes of the interval in reverse postorder.
$                 Iterating over INT_NODES(I) is equivalent to iterating
$                 forward over the nodes in I.

$ 4. VEDGES:      The set of all edges which are part of some higher
$                 order graph.

$ 5. PROPER_INTS: A set of all proper (reducible) intervals.



$ STEP 1: Calculate the following objects:

$ 1. NODENO:      Maps each node into its preorder index
$ 2. POSTNO:      Maps each node into its postorder index
$ 3. NDESCS:      Maps each node into the number of its descendants
$ 4. NODES:       Tuple of nodes in preorder.
$ 5. POSTNODES:   Tuple of nodes in postorder
$ 6. BACKINV:     The set of all [Y, X] such that [X, Y] is a back edge
$ 7. TARGBACK:    A tuple of targets of back edges in preorder.

$ (1) - (5) are built by an auxiliary depth-first searching routine
$ 'DFST'.  When we build the node indices we use only even numbers.
$ This leaves the odd numbers for target blocks (i.e. interval
$ preheaders). Initially only the even elements of NODES and POSTNODES
$ are filled in.
```

```
$ The following macro tests for tree descendancy.

    MACRO IS_DESC(X, Y);
        (NODENO(Y) <= NODENO(X) AND NODENO(X) <= NODENO(Y)+NDESCS(Y))
    ENDM;

    DFST(R);    $ Construct a depth-first spanning tree.

$ Construct the set BACKINV of all reverse back edges
    BACKINV := { [Y, X] IN PRED ST ROUTOF(Y) = R AND IS_DESC(X, Y)};

$ Construct the tuple TARGBACK of all back edge target nodes, arranged
$ in reverse preorder.
    TARGBACK := [NODES(I): I := # NODES, # NODES-1 ... 1 ST
        NODES(I) /= OM AND NODES(I) IN DOMAIN BACKINV];

$ STEP 2

$ At this point 'TARGBACK' contains all potential interval heads in
$ reverse preorder. We iterate over X in TARGBACK doing three things:

$ 1. Build the set 'IMPROPERS' of such nodes X which are heads of
$    multiple-entry loops, and thus are 'sources of irreducibility'.

$ 2. For each X find the set 'REACHUNDER' of nodes (in the reduced
$    graph in which each already processed proper or improper
$    interval has been logically 'squashed', i.e. identified with
$    a single node - its target block) which reach X along a path
$    not passing through X whose final edge is a back edge. If any
$    node which is not a descendant of X belongs to 'REACHUNDER',
$    then X is a head of a multiple-entry loop, and we add X to
$    'IMPROPERS'. Otherwise X is a head of a single-entry loop,
$    and thus is an interval head in our sense; if
$    REACHUNDER * IMPROPERS = {}, then that interval is a proper
$    interval, and we add it to 'PROPER_INTS'; otherwise it is an
$    improper interval.

$ 3. If X is an interval head then:

$    a. Create a new target block 'TBX'.
$    b. Add TBX to 'INTS(R)' and set INT_NODES(TBX) to [].
$    c. For all Y in REACHUNDER, set INTOF(Y) := TBX
$    d. Update the flow graph to show the insertion of TBX.

    ROOT := RENTRY(R);

    INTS(R) := [];

    (FORALL X IN TARGBACK)
        REACHUNDER    := {X};
        NEWREACHUNDER := { INTOF .LIM Y : Y IN BACKINV{X}} - {X};
$ INTOF .LIM Y is the largest interval constructed so far which
$ contains Y (see below for details).
```

```
        (WHILE NEWREACHUNDER /= {})
            Y FROM NEWREACHUNDER;
            REACHUNDER WITH Y;  $ Get a new element of 'REACHUNDER'

            IF NOT IS_DESC(Y, X) THEN  $ We have a multiple-entry loop
                IMPROPERS WITH X;
                QUIT WHILE;            $ Exit the while loop
            ELSE
                NEWREACHUNDER +:=
                    {[ INTOF .LIM Z : Z IN PRED{Y}} - REACHUNDER);
            END IF;
        END WHILE;

        IF X IN IMPROPERS THEN CONTINUE FORALL; END;

$ Here X is an interval head.

        TBX := GET_TARG(X);
$ The GET_TARG routine creates a new basic block, initially containing
$ only a label and an unconditional jump to X.  Code for this routine
$ is omitted here.

$ Insert TBX in proper place in the tree, and initialize its attributes
        NODENO(TBX) := NODENO()-1;
        NODES(NODENO(TBX)) := TBX;
        POSTNO(TBX) := POSTNO(X)+1;
        POSTNODES(POSTNO(TBX)) := TBX;
$ Note that there is no need to compute NDESCS(TBX), as this value will
$ not be used later.

        INT_NODES(TBX) := [];

$ TBX represents the interval with head X.

        INTS(R) WITH TBX;

$ Check if TBX is proper

        IF REACHUNDER * IMPROPERS = {} THEN
            PROPER_INTS WITH TBX;
        END IF;

$ Map each node in REACHUNDER to its containing interval TBX
        (FORALL Y IN REACHUNDER) INTOF(Y) := TBX; END;

$ Update the flow graph to account for the insertion of TBX into it.
$ This involves the following actions:

$ 1. Add an edge [TBX,X] to the graph.
```

```
$ 2. Replace all edges entering the interval through 'X' by edges
$    entering TBX, and change the corresponding branch instructions
$    in the program code.

$ 3. For each edge [U,V] leaving the interval whose head is X, add
$    a 'virtual' edge [TBX,V] to the graph.  This edge is added to
$    'VEDGES'.

        UPDATE(X, TBX, REACHUNDER);
    END FORALL;

$ Build the outermost 'interval', identified by the entry node 'ROOT'.

    INTS(R) WITH ROOT;
    INT_NODES(ROOT) := [];
    PROPER_INTS WITH ROOT;    $ Root will be removed from this set if
                              $ actually improper

$ Iterate over the nodes in reverse postorder, adding each node to
$ INT_NODES.  If a node has its interval head undefined put it in the
$ outermost interval.

    (FOR I := # POSTNODES, #POSTNODES-1 ... 1)
        X := POSTNODES(I);
        IF X = OM THEN CONT FOR I; END;

        HD := INTOF(X);
        IF HD = OM THEN
            HD := INTOF(X) := ROOT;
            IF X IN IMPROPERS THEN
                PROPER_INTS LESS ROOT;
            END IF;
        END IF;

        INT_NODES(HD) WITH X;
    END FOR;

    END PROC FIND_INTS;
```

```
    PROC UPDATE(X, TBX, INODES);

$ This routine updates the flow graph to show the insertion of
$ the target block 'TBX'. Its arguments are:

$ X:        The interval head
$ TBX:      The target block
$ INODES:   The nodes in the interval

$ In this code, only manipulation of the flow graph is shown; code
$ manipulating individual instructions within blocks is omitted.

    CESSOR(TBX) WITH X;
    PRED(X)     WITH TBX;

$ Next we iterate over all the predecessors of X which are not in
$ the interval modifying the CESSOR and PRED maps as we go.


    (FORALL Y IN PRED(X) ST Y NOTIN INODES + (TBX))

        CESSOR(Y)  LESS X;
        CESSOR(Y)  WITH TBX;

        PRED(X)   LESS Y;
        PRED(TBX) WITH Y;
    END FORALL;


$ Find all edges which leave the interval and add a virtual edge
$ from TBX for each such edge.

    (FORALL U IN INODES, Y IN CESSOR(U)
     ST Y NOTIN INODES AND INTOF(Y) /= U)
        CESSOR(TBX) WITH Y;
        PRED(Y)     WITH TBX;

        VEDGES WITH [TBX, Y];
    END FORALL;

    END PROC UPDATE;
```

```
    PROC DFST(R);

$ This routine builds the depth first spanning tree for a routine
$ 'R'. We initialize counters for the various node indices and then
$ call 'DFST1' to do the recursive tree walk.

    NODENO := {};
    POSTNO := {};
    NDESCS := {};
    NODES := [];
    POSTNODES := [];
    SEEN := {};
    NPRE := NPOST := 0;
    DFST1(RENTRY(R));

    END PROC DFST;
```

```
    PROC DFST1(X);

$ This routine builds the depth first spanning subtree rooted at the
$ node 'X'.

    NODENO(X) := (NPRE +:= 2);    $ Note the use of even indices only
    NDESCS(X) := 0;

    NODES(NPRE) := X;
    SEEN WITH X;

    (FORALL Y IN CESSOR(X) ST Y NOTIN SEEN)
        DFST1(Y);
        NDESCS(X) +:= (NDESCS(Y) + 2);
$ Each node is counted as two descendants, to match the usage of
$ only even indices in NODENO and POSTNO.
    END FORALL;

    POSTNO(X) := (NPOST +:= 2);
    POSTNODES(NPOST) := X;

    END PROC DFST1;
```

```
    OP .LIM(F, X);

$ This operator finds a value 'Y' such that Y = F(F(F ... F(X))))
$ and F(Y) = OM.

$ Note that unlike Tarjan's original approach we omit path
$ compression, tree balancing, etc. for the sake of simplicity,
$ though these could easily be added.

    Y := X;
    (WHILE F(Y) /= OM) Y := F(Y); END;

    RETURN Y;

    END OP .LIM;
```

```
    MODULE SETL_OPTIMIZER - DATAFLOW_SOLVER;

$ This module contains a package of general purpose
$ routines to solve bit vector data flow problems either
$ intraprocedurally or interprocedurally. We can distinguish
$ between four basic types of such analyses, according to
$ the character of the desired analysis:

$ FORWARD  - Data is to be propagated in the direction of
$            the flow, from procedure entries forward.

$ BACKWARD - Data is to be propagated in the reverse
$            direction of the flow, from exits backward.

$ MEET     - Whenever two paths converge (for forward analysis)
$            or diverge (for backward analysis) take the meet (set
$            intersection) of data values propagated along these
$            paths.

$ JOIN     - As in MEET, except that the join (set union) of the
$            corresponding data values is to be taken.

$ Typical examples are: expression availability analysis
$ is a forward - meet analysis; unconditional exposure
$ of expressions (also known as 'very busy' expressions
$ analysis) is a backward - meet analysis; reaching
$ definitions analysis is a forward - join analysis, and
$ live variables analysis is a backward - join analysis.

$ As noted in chapters 5 and 6, forward and backward analyses
$ require substantially different logic, so that each of them
$ is executed in a different subpackage; however, the
$ difference between meet and join problems turns out to
$ be rather minor, so that they both can be handled by
$ the same (forward or backward) package, using a switch
$ to indicate whether a particular analysis is of meet or
$ join type.

$ This module exports the following procedures:

$ CGRAPH_ANALYSIS - Call graph analysis routine, to be called
$                   once before solving any data flow problem
$                   interprocedurally.

$ INTERPROC_FWD_ANALYSIS - Call this to solve interprocedural
$                   forward data flow analyses.

$ INTRAPROC_FWD_ANALYSIS - Call this to solve intraprocedural
$                   forward data flow analysis for a given
$                   procedure.
```

```
$ INTERPROC_BACK_ANALYSIS - Call this to solve interprocedural
$                   backward data flow analysis

$ INTRAPROC_BACK_ANALYSIS - Performs intraprocedural backward
$                   analysis for a given procedure.

$ This package assumes the following global objects to be
$ available:

$ CGRAPH     - The program call graph, represented as a set
$              of edges; an edge (P,Q) is in CGRAPH iff P is a
$              procedure which contains a call to the procedure Q.

$ ROUTS      - Set of all program procedures (i.e. all nodes
$              of the call graph).

$ SYM_MAIN   - Main-program identifier (i.e. the entry node of the
$              call graph).

$ ROUTOF     - Maps each block to the procedure containing it.

$ RENTRY     - Maps each procedure to its entry block.

$ REXIT      - maps each procedure to its exit (return) block.

$ RSTOP      - Maps each procedure to its stop block, if any.

$ CALLSIN    - Maps each procedure to the set of all call blocks
$              in it.

$ CALLPROC   - Maps each call block to the procedure it calls.

$ CESSOR     - The program flow graph, as a union of the flow
$              graphs of all procedures. An edge (M, N) is in
$              CESSOR iff either M contains a branch to N, or else
$              M is a call block and N is the block immediately
$              following N. The nodes of the flow graph are either
$              basic blocks or derived intervals (which are
$              represented by their target blocks), in which case
$              an edge (INT, V) in CESSOR can indicate the possibility
$              of a transfer of control from the interval INT to a
$              successor V of some node in INT. These edges are called
$              virtual edges (as above; see the interval
$              analysis package for more details).

$ PRED       - The inverse map of CESSOR.

$ INTS       - Maps each procedure to the tuple of its intervals
$              in reverse preorder (relative to a depth first
$              spanning tree of its flow graph).
```

```
$ INT_NODES - Maps each interval to the sequence of its nodes
$             in interval order (i.e., reverse postorder).

$ PROPER_INTS - The set of all proper intervals (those which do
$             not contain irreducible nuclei).

$ INTOF     - Maps each flow graph node to the interval containing
$             it.

$ VEDGES    - Set of all virtual edges (see the description of
$             CESSOR above).


$ In addition this module uses the following global-within-
$ the-module variables, the first three of which are used
$ to transmit flags and analysis constants between inner routines,
$ while the rest are built by a recursive depth-first search
$ procedure during call-graph analysis, and are used later in that
$ analysis.

    VAR
        ID,         $ Identity flow map
        ZERO,       $ Null data state
        MEET_FLAG,  $ TRUE if meet analysis; otherwise FALSE
        SEEN,       $ Procedures already in DFST of cgraph
        CNPRE,      $ Current preorder index in DFST
        CNPOST,     $ Current postorder index in DFST
        NODENO,     $ preorder numbering map
        POSTNO,     $ postorder numbering map
        NDESCS;     $ No. of descendants map
```

```
    PROC CGRAPH_ANALYSIS;

$ This procedure performs the call graph analysis needed for
$ our interprocedural data flow analysis solver. It computes
$ the following objects:

$ CG_SCCS     - A tuple of (roots of the) strongly connected
$               components of cgraph, arranged in reverse postorder.

$ SCC_NODES   - Maps each (root of a) strongly connected component
$               into a tuple containing its nodes in reverse
$               postorder.

$ SCC_D       - Maps each (root of a) strongly connected component
$               S into an estimate of its loop-interconnectedness
$               parameter D, defined as the maximal number of back
$               edges along any acyclic path in S (we do not attempt to
$               obtain that precise value, but rather use a crude
$               upper bound for it, namely the number of back
$               edge targets contained in S.)


$ Begin by calling a standard depth first spanning tree
$ routine, which will compute the following objects:

$ NODENO - Preorder node numbering map.
$ POSTNO - Postorder node numbering map.
$ NDESCS - Number of descendants map.

    CDFST();

$ Tree-descendancy macro, identical to the one used for interval
$ analysis.
    MACRO IS_DESC(P, Q);  $ Test whether P is a descendant of Q
        (NODENO(P) >= NODENO(Q) AND NODENO(P) <= NODENO(Q)+NDESCS(Q))
    ENDM;

$ Next compute some auxiliary objects:

    INVERSE := {[P, Q] : [Q, P] IN CGRAPH}; $ Inverse call graph
    INVPOSTNODES := {[#ROUTS+1-N, P] : N := POSTNO(P)};
              $ Procedures in their reverse postorder
    BACKINV := {[P, Q] IN INVERSE ST IS_DESC(Q, P)};
              $ Set of all inverse back edges
    TARGBACK := DOMAIN BACKINV;   $ Back edge targets

    CG_SCCS := [];      $ See above
    SCC_NODES := SCC_D := {};    $ See above

    SCCROOT := {};  $ Strongly connected component root map
```

```
$ Iterate through the procedures, looking for strongly
$ connected components.

    (FOR I := 1 ... #INVPOSTNODES)
        P := INVPOSTNODES(I);
        IF SCCROOT(P) = OM THEN  $ We have a new root of a S.C.C.
            SCCROOT(P) := P;
            CG_SCCS WITH P;   $ P corresponds to the new component
            SCC_NODES(P) := [P];
            IF P IN TARGBACK THEN  $ This is a non-trivial S.C.C.
                NEWNODES := BACK(INV{P} - {P};
                      $ New nodes to be added to the S.C.C.
                SCC_D(P) := 1;
              $ develop count of no. of backedge targets in the S.C.C.
                (WHILE NEWNODES /= {})
                    Q FROM NEWNODES;
                    SCCROOT(Q) := P; $ Mark Q as belonging to the SCC
                    IF Q IN TARGBACK THEN SCC_D(P) +:= 1; END;
                    NEWNODES +:= {R IN INVERSE{Q} ST
                                  IS_DESC(R, P) AND SCCROOT(R) = OM};
                END WHILE;

            ELSE         $ We have a trivial S.C.C.
                SCC_D(P) := 0;
            END IF;

        ELSE       $ P belongs to a SCC already scanned
            SCC_NODES(SCCROOT(P)) WITH P;
        END IF;
    END FOR;

    PRINT(' ');
    PRINT('   C A L L    G R A P H    A N A L Y S I S');
    PRINT(' ');
    PRINT('CG_SCCS =', CG_SCCS);
    PRINT('SCC_NODES =', SCC_NODES);
    PRINT('SCC_D =', SCC_D);
    END PROC CGRAPH_ANALYSIS;
```

```
    PROC CDFST;

$ This routine builds the depth first spanning tree of the call
$ graph. We initialize counters for the various node numberings
$ and then call 'CDFST1' to do the recursive tree walk.

    NODENO := NODESCS := POSTNO := {};
    SEEN := {};
    CNPRE := CNPOST := 0;
    CDFST1(SYM_MAIN);

    END PROC CDFST;
```

```
    PROC CDFST1(P);

$ This routine builds the depth first spanning tree starting with
$ node 'P'. This routine differs in various details from the depth
$ first spanning routine used for interval analysis.

    NODENO(P) := (CNPRE +:= 1);
    NDESCS(P) := 0;

    SEEN WITH P;

    (FORALL Q IN CGRAPH(P) ST Q NOTIN SEEN)
        CDFST1(Q);
        NDESCS(P) +:= (NDESCS(Q) + 1);
    END FORALL;

    POSTNO(P) := (CNPOST +:= 1);

    END PROC CDFST1;
```

```
    PROC INTERPROC_FWD_ANALYSIS(RW F, WR SOLN, ID_PRM, ZERO_PRM,
                                MEET_FLAG_PRM, MOVE_CODE,
                                RW EXPOSED, WR INSERT);

$ Note declarations of 'read-write' parameters ('RW') and 'write-only'
$ parameters ('WR').

$ This is the master routine to perform a specific data flow
$ analysis interprocedurally. Its parameters are:

$ F    - Maps each edge (M, N) in the flow graph to a compact
$        representation of its data-propagation map F(M,N).
$        Initially this information has to be provided only
$        for basic blocks (but not for call blocks); the
$        first phase of the analysis will fill
$        in additional entries. Each F(M,N) is represented
$        as a pair [A, B] in L x L, such that for each X in L
$        F(M,N)(X) = X*A + B, and A contains B (this latter
$        condition ensures that the representation is unique,
$        and also simplifies some functional manipulations).

$ SOLN- The solution vector for the analysis. SOLN maps each
$       flow graph node to the data found to be known at its
$       entry.

$ The next three parameters are transmitted internally between
$ subprocedures by assigning them to global variables, as they
$ are constant per analysis. The corresponding globals are:

$ ID   - The identity map representation. ID = [U, {}], where
$        U is the universal set over which bitvectors are taken
$        in this analysis (e.g. set of all program expressions,
$        set of all variables etc.)

$ ZERO - The initial data value, i.e. flow data assumed at the main
$        program entry.

$ MEET_FLAG - A flag indicating whether the analysis is a meet
$        analysis or a join analysis.

$ AUX_F - These are auxiliary propagation maps. For each flow
$        graph node U, AUX_F(U) denotes the effect of propagation
$        from the entry to I, the interval containing J, through
$        I, to the entry of U.

$ MOVE_CODE - A flag indicating that code motion is required.
```

```
$ EXPOSED - This is initially the set of computations (corresponding
$       to analysis elements (bits)) exposed at the start of each
$       basic block N (i.e. computed with no prior kill in N). The
$       inner-to-outer phase of our analysis attaches an 'EXPOSED'
$       value to each interval processed. EXPOSED(I) is the set of all
$       expressions T for which there exists a computation of T
$       within the interval I which would become redundant if and
$       only if T became available at the entry to (the target block
$       of) I. Note, however, that the logical place at which
$       computations movable out of an interval I should be inserted
$       is the end of the target block of I, rather than its start.
$       Thus if that target block is nonempty then EXPOSED(I)
$       need not represent those movable computations. For this
$       reason we provide the parameter 'INSERT' which gives the
$       desired set of movable code.

$ INSERT - This output parameter will map each interval into
$       the set of all computations movable out of its loop,
$       which are to be inserted at the end of the target
$       block of the interval. The actual insertion should be
$       performed by the calling procedure.


$ Our analysis procedures  make frequent use of the following
$ operators (which could be also written as macros, if it were
$ not for the convenience of the infix notation that we prefer
$ to use):

$ .COMP      - Functional composition
$ .MEETJOIN - Functional meet or join, depending on MEET_FLAG
$ .MJV       - Meet or join of lattice values
$ .OF        - Functional application

$ All these operators have elementary set expressions; see below
$ for details.

$ Note also that these operators must be prepared to handle
$ undefined flow values, which will be represented
$ by a special constant 'FOM'; for example,
$ G .COMP FOM = FOM .COMP G = FOM;
$ (concatenation of an undefined flow with a defined one is
$ still undefined)
$ G .MEETJOIN FOM = FOM .MEETJOIN G = G.
$ (a join or a meet of an undefined flow with a defined flow
$ yields the defined flow.)

$ another special constant 'XOM' is used to denote the undefined data
$ state in L.
```

```
$ Transfer constant parameters to globals

    ID := ID_PRM;
    ZERO := ZERO_PRM;
    MEET_FLAG := MEET_FLAG_PRM;

$ The master procedure consists of the following three phases:

$ Interprocedural elimination phase

    AUX_F := INTERPROC_FWD_ELIMINATE(F);

$ If code motion is required then perform an additional
$ phase, computing the sets of movable code.

    IF MOVE_CODE THEN
        INSERT := {};
        (FORALL P IN ROUTS)
            PROPAGATE_EXPOSED(P, F, AUX_F, EXPOSED, INSERT);
        END FORALL P;
    END IF;

$ Find data at procedure entries

    ENT_INF := ENTRY_INFO(F, AUX_F, INSERT);

$ Final propagation phase

    SOLN := {};      $ Initialize the solution
    (FORALL P IN ROUTS)
        FWD_PROPAGATE_IN(P, F, AUX_F, SOLN, ENT_INF(P),
                         MOVE_CODE, INSERT);
    END FORALL;

    RETURN;

    END PROC INTERPROC_FWD_ANALYSIS;
```

```
    PROC INTERPROC_FWD_ELIMINATE(RW F);

 $ This is the driver routine for the first interprocedural
 $ inner-to-outer interval pass. Procedures are analyzed in
 $ the following order: we process the strongly connected
 $ components of the call graph in their postorder; for each
 $ such component, we iterate through its procedures in their
 $ postorder, no more than 2*D+1 times, where D is the loop-
 $ interconnectedness parameter of the component.

    AUX_F := {};    $ Initialize auxiliary maps

$ Iterate through the S.C.C.s of cgraph
    (FOR I := #CG_SCCS, #CG_SCCS-1 ... 1)

        SCC := CG_SCCS(I);    $ get a S.C.C.
        SCC_PROCS := SCC_NODES(SCC);  $ Procs in that S.C.C.
        FLOW_FLAG := 'FIRST_INTER'; $ First processing of SCC

        (FOR J := 1 ... 2 * SCC_D(SCC) + 1 UNTIL PROC_CONVERGE)

            PROC_CONVERGE := TRUE;

            (FOR K := #SCC_PROCS, #SCC_PROCS-1 ... 1)

                P := SCC_PROCS(K);
                PROC_CONVERGE :=
                    INTRAPROC_FWD_ELIMINATE(P, AUX_F, F, FLOW_FLAG)
                        AND PROC_CONVERGE;
 $ The INTRAPROC_FWD_ELIMINATE routine analyzes P; its fourth parameter
 $ indicates whether the analysis is first-time interprocedural, second
 $ -time interprocedural or intraprocedural; it returns a flag to
 $ indicate whether information in P has stabilized.

            END FOR K;

            FLOW_FLAG := 'SECOND_INTER'; $ Additional passes thru SCC

        END FOR J;

    END FOR I;

    RETURN AUX_F;
    END PROC INTERPROC_FWD_ELIMINATE;
```

```
PROC INTRAPROC_FWD_ELIMINATE(P, RW AUX_F, RW F, FLOW_FLAG);

$ This routine performs an intraprocedural elimination phase
$ for the procedure P, using interval analysis. The fourth parameter
$ indicates whether this routine has been invoked by the
$ intraprocedural solver or by the interprocedural solver, and
$ in the second case, whether this is the first time P is
$ being processed or not.

$ In this pass we iterate through the procedure's intervals
$ in an inner-to -outer order (i.e. in reverse preorder of their
$ heads in a DFST of the flow graph of P). For each interval
$ I processed in this manner we compute a set of data-propagation
$ maps of the form F(I, J), where

$ (1) If U is in I, then this map is an auxiliary map (which will
$ be denoted as AUX_F(J), I being implicit in this case) which
$ represents the propagation effect as control advances from
$ the start of I, thru I, to the start of U;

$ (2) If U is not in I, then J is a successor of some node in I.
$ Here the map F(I, U) represents the propagation effect as control
$ advances from the start of I, through I, to the start of U;
$ in this case F(I, U) is needed for the processing of the
$ intervals containing I. Note that [I, U] is a virtual edge
$ in our flow graph; thus the elimination phase extends the
$ map F so as to be defined also on virtual edges.

$ Any interval I processed in this routine is either a proper
$ strongly connected interval, or, if it contains 'improper'
$ nodes (i.e. nuclei of irreducibility), is a single-entry
$ strongly connected subgraph. In the first case we only have to
$ iterate thru the nodes of I twice, but in the second case till
$ convergence.

$ The outermost 'interval' is either a single entry acyclic
$ graph (if it does not contain irreducible nuclei), or a
$ general single-entry graph otherwise. For this 'interval' we
$ iterate either once in the first case, or till convergence
$ otherwise.

$ If the present routine is to be used for interprocedural analysis,
$ we first reset the propagation maps for call blocks in P. If none of
$ these maps have changed from the last processing of P,
$ then obviously analysis of P has stabilized and we can return
$ immediately. Moreover, intervals need be re-processed if and only
$ if they contain a call block whose local effect has changed,
$ or, recursively, contain an interval whose local effects
$ have changed. In terms of the 'INTOF' tree, we only have to
$ re-analyze intervals lying along some path from the
$ root to a call block whose local effect has changed. This
$ can make reprocessing of a procedure considerably
$ faster than initial processing.
```

```
    IF FLOW_FLAG = 'SECOND_INTER' THEN
$ Process only intervals containing calls with new effect
        NEED_PROCESS := {};
    ELSE
$ Process all intervals
        NEED_PROCESS := { INTT : INTT IN INTS(P)};
    END IF;

    IF FLOW_FLAG /= 'INTRA' THEN
$ Interprocedural analysis.

        (FORALL C IN CALLSIN(P))

            V := CESSOR(C);      $ The block following the call
            P1 := CALLPROC(C);   $ C calls P1
            EP1 := REXIT(P1);    $ The return block of P1.
$ (Note here that if this routine is modified to include parameter-
$ passing assignments as part of call blocks, in the manner suggested
$ in a concluding remark in SECTION 4, then one might manipulate
$ AUX_F(EP1), which defines the local effect of executing P1, to get
$ F(C,V), rather than just assign the first map to the second one, as
$ is done below).

            IF F(C, V]) /= AUX_F(EP1) THEN

$ Update flow function for call
                F(C, V]) := IF AUX_F(EP1) = OM THEN FOM
                            ELSE AUX_F(EP1) END;

$ Interval containing call must be processed
                NEED_PROCESS WITH INTOF(C);
            END IF;
        END FORALL C;

$ If no intervals need be processed then information has
$ stabilized and no re-processing of P need be done.

        IF NEED_PROCESS = {} THEN RETURN TRUE; END;

    END IF;

    P_INTS := INTS(P); $ Intervals of P in reverse preorder
    OUTINT := P_INTS(#P_INTS);  $ Outermost interval

    (FORALL INTT := P_INTS(1) ST INTT IN NEED_PROCESS)

        NEED_PROCESS WITH INTOF(INTT);  $ Process containing interval
        NODES := INT_NODES(INTT); $ Nodes of INTT in interval order

        HEAD := NODES(1);  $ Interval head
        AUX_F(HEAD) := ID;  $ Initialize AUX_F of HEAD to the identity
```

```
$ Note here that the edge [INTT, HEAD] is a real edge in the
$ flow graph, so that F([INTT, HEAD]) will have been pre-computed in
$ an initialization phase, along with the flow maps for all other
$ real edges, and is therefore available here.

$ Three cases are now possible:
$ (1) INTT is proper, but not outermost; then iterate twice.
$ (2) INTT is proper, and is outermost; then iterate once.
$ (3) INTT is improper; iterate indefinitely (1 + number of
$     nodes is an adequate upper bound) until convergence.
$     (Note that we do not make use of the better upper bound on
$     the number of iterations discussed in SECTION 3).

        CONV_CONTROL := INTT NOTIN PROPER_INTS;
$ Test for convergence only in this case

        N_ITER :=       $ Maximal number of iterations
            IF INTT NOTIN PROPER_INTS THEN #NODES + 1
            ELSEIF INTT = OUTINT THEN 1 ELSE 2 END;

$ If improper interval, initialize AUX_F of all non-head nodes
$ to 'FOM'. This is because we cannot guarantee in this case that
$ when propagating data to a node within INTT, all its predecessors
$ (within INTT) have already been processed, so that we have to
$ prepare for the case where some of these predecessors still
$ have undefined auxiliary data-flow maps.

        IF CONV_CONTROL THEN
            (FOR J := 2 ... #NODES)
                AUX_F(NODES(J)) := FOM;
            END FOR;
        END IF;
$ Iterate through nodes of INTT.

        (FOR D := 1 ... N_ITER UNTIL CONVRGD)

            CONVRGD := CONV_CONTROL;

$ Iterate thru nodes of INTT, other than HEAD

            (FOR J := 2 ... #NODES)

                ND := NODES(J);
                FTEMP := .MEETJOIN/
                    {F([PND,ND]) .COMP AUX_F(PND) : PND IN PRED(ND)
                            ST INTOF(PND) = INTT};
                PRINT('AUX_F(',ND,') =',FTEMP);
                CONVRGD := CONVRGD AND (FTEMP = AUX_F(ND));
                AUX_F(ND) := FTEMP;

            END FOR J;

$ Test if processing of INTT has terminated
            IF D = N_ITER OR CONVRGD THEN QUIT FOR D; END;
```

```
$ Re-compute AUX_F(HEAD), taking back edges into account
            FTEMP := .MEETJOIN/ {F([PND, HEAD]) .COMP AUX_F(PND) :
                        PND IN PRED(HEAD) ST INTOF(PND) = INTT};
               $ Note that a meet/join over an empty set yields OM
            FTEMP := IF FTEMP = OM THEN AUX_F(HEAD)
                     ELSE AUX_F(HEAD) .MEETJOIN FTEMP END;
            IF NOT CONV_CONTROL THEN
                CONVRGD := (AUX_F(HEAD) = FTEMP);
            END IF;

            AUX_F(HEAD) := FTEMP;
        END FOR D;


$ Compute F([INTT, V]), where V is a successor of some node in
$ INTT; note that this loop will be null for the
$ outermost interval.

        (FORALL V IN VEDGES(INTT))
            FTEMP := .MEETJOIN/ {F([PV, V]) .COMP AUX_F(PV) :
                           PV IN PRED(V) ST INTOF(PV) = INTT};
            F([INTT, V]) := FTEMP .COMP F([INTT, HEAD]);

        END FORALL V;

    END FORALL INTT;

    RETURN FALSE;   $ To indicate no convergence.

    END PROC INTRAPROC_FWD_ELIMINATE;
```

```
    PROC PROPAGATE_EXPOSED(P, RW F, AUX_F, RW EXPOSED, RW INSERT);

$ This procedure performs an inner-to-outer pass over all
$ intervals to determine the computations which might be moved
$ out of the loop of each interval I. As explained above,
$ these computations are not necessarily those exposed in I;
$ hence, we build up both sets 'EXPOSED' and 'INSERT'
$ simultaneously.

$ In this analysis, the set of computations movable out of the
$ loop of I is obtained by taking all computations T with
$ the property that there exists a node ND in I such that
$ T is exposed in ND and is available at the start of ND iff
$ it is available at the end of the target block of I.

$ The movable code is always assumed to be appended to the
$ end of the target block of the interval, to avoid any possible
$ conflict with code that is already present in the target block.
$ However, this appending takes place physically only at the end
$ of the elimination phase. Thus, we do not attempt to make
$ use of the fact that these expressions are potentially
$ available at the head of I in updating any flow function.
$ This approach is necessary to ensure convergence of our algorithms
$ in cases of recursive cycles of interprocedural flow.

    P_INTS := INTS(P);  $ Intervals of P in reverse preorder

$ First extend F to indicate null flow from the entry block to
$ itself. Since the outermost interval has no target block,
$ and is therefore identified with its head, this trick unifies
$ the treatment of that interval with the treatment of inner
$ intervals, as shown below.

    OUTINT := P_INTS(#P_INTS);
    F([OUTINT, OUTINT]) := ID;

    (FORALL INTT := P_INTS(<))

        NODES := INT_NODES(INTT);
        HEAD := NODES(1);

$ In computing EXPOSED(INTT), we must reckon with the fact
$ that the target block of INTT (also denoted by INTT)
$ might be non-empty, due to prior code motion. This can mean that
$ (a) F([INTT, HEAD]) is not the identity, and (b) EXPOSED(INTT)
$ (where INTT is treated as a basic block) is not null
$ initially.
```

```
$ We proceed as follows: first find all exposed computations in
$ the loop of INTT, assuming the target block of INTT to be null.
$ These are the computations movable out of the loop of INTT.

        INSERT(INTT) := +/ [EXPOSED(ND) *
                    (AUX_F(ND)(1) - AUX_F(ND)(2)) : ND IN NODES];

$ Next find the new set of computations which are still exposed
$ at the entry to the target block of INTT.

        FTARG := F([INTT, HEAD]);
        EXPFROMENTRY := INSERT(INTT) * (FTARG(1) - FTARG(2));

$ Add these computations to those exposed in the target block
        EXPOSED(INTT) := EXPOSED(INTT) + EXPFROMENTRY;


    END FORALL INTT;

    RETURN;

    END PROC PROPAGATE_EXPOSED;
```

```
    PROC ENTRY_INFO(F, AUX_F, INSERT);

$ This function calculates and returns a mapping which sends
$ each procedure P into the flow information available at entry
$ to P. It is called (only in the interprocedural case) just
$ before we begin the final outer-to-inner propagation phase.

$ First we construct a map 'CGF' assigning to each edge (P, Q)
$ of the call graph a data-propagation map, describing the
$ propagation effect as control advances from the entry of P
$ to the entry of Q via any call to Q from P.

    CGF := {};
    (FORALL [P,Q] IN CGRAPH) CGF([P,Q]) := FOM; END;

    (FORALL Q := CALLPROC(C))  $ For all calls within all procedures

        P := ROUTOF(C);   $ [P, Q] is an edge of the call graph

$ Compute the local effect as control advances from the entry
$ of P to C.

        FTEMP := AUX_F(C);

        (INIT IU := INTOF(C); WHILE IU /= RENTRY(P))

            HIU := INT_NODES(IU)(1);   $ head of IU
$ Add the effect of code moved out of IU
            FINS := [ID(1), INSERT(IU)];
            FTEMP := FTEMP .COMP FINS .COMP F([IU, HIU])
                                 .COMP AUX_F(IU);

            IU := INTOF(IU);

        END;

        CGF([P, Q]) := CGF([P, Q]) .MEETJOIN FTEMP;

    END FORALL Q;

$ Next we iterate through the call graph in 'invocation order', i.e.
$ process the strongly connected components in reverse postorder
$ and the set of procedures within each strongly connected
$ component in reverse postorder also.

    ENT_INF := {[P, NOM] : P IN ROUTS};  $ Initialize solution
    ENT_INF(SYM_MAIN) := ZERO;
    CGRINV := {[P, Q] : [Q, P] IN CGRAPH};
```

```
    (FOR I := 2 ... #CG_SCCS)   $ Pick S.C.C.'s in reverse postorder.

$ Note that we assume here that the main program is non-recursive,
$ so that the first strongly-connected component of the call
$ graph consists of the main program only. Thus we can skip it,
$ for the entry value of the main program is already assumed
$ known.

        SCC := CG_SCCS(I);
        SCC_PROCS := SCC_NODES(SCC); $ Procs in SCC in rev. postorder

        (FOR N := 1 ... SCC_D(SCC) + 1 UNTIL CONVRGD)

            CONVRGD := TRUE;

            (FORALL P := SCC_PROCS(K))

                TEMP := +/// {CGF([Q, P]) .OF ENT_INF(Q) :
                        Q IN CGRINV(P)};
$ Test for convergence
                CONVRGD := CONVRGD AND (TEMP = ENT_INF(P));
                ENT_INF(P) := TEMP;

            END FORALL P;

        END FOR N;

    END FOR I;

    RETURN ENT_INF;
    END PROC ENTRY_INFO;
```

```
    PROC FWD_PROPAGATE_IN(P, RW F, AUX_F, RW SOLN, ENT_VAL,
                          MOVE_CODE, RW INSERT);

$ This procedure performs outer-to-inner propagation for a
$ routine P, using the 'interval-effect' flow functions AUX_F
$ to modify the solution map 'SOLN'. The parameter ENT_VAL
$ gives the flow information assumed (or known) at procedure
$ entry.

$ If code motion is required, then the computations in INSERT(I)
$ are assumed to be available at the end of the target block
$ of an interval I (but only for the purpose of propagation
$ inside I). In addition, computations in INSERT(I) already
$ available at exit from the target block of I are removed from
$ INSERT(I).

$ Note that movable computations are assumed to be such that the
$ insertion of any of them will not 'kill' any others.

    SOLN(RENTRY(P)) := ENT_VAL;
    P_INTS := INTS(P);  $ Intervals of P in reverse preorder

$ Extend F to indicate null flow from the entry block to
$ itself. Since the outermost interval has no target block,
$ and is therefore identified with its head, this trick unifies
$ the treatment of that interval with the treatment of inner
$ intervals, as shown below.

    OUTINT := P_INTS(#P_INTS);
    F([OUTINT, OUTINT]) := ID;

    (FOR K := #P_INTS, #P_INTS-1 ... 1)

        INTT := P_INTS(K);
        NODES := INT_NODES(INTT);  $ nodes of INTT

        SOLN1 := SOLN(INTT);  $ Data value at entry to INTT
$ Convert SOLN1 to the data attribute value at the end of the target
$ block of INTT.

$ Propagate through the target block of INTT; if INTT = OUTINT, the
$ trick noted above will make the following statement a no-op.

        SOLN1 := F([INTT, NODES(1)]) .OF SOLN1;

$ If code motion is also required, then update INSERT(INTT)
$ and add it to SOLN1.
        IF MOVE_CODE AND INTT /= OUTINT THEN
            INSERT(INTT) := INSERT(INTT) - SOLN1;
            SOLN1 := SOLN1 + INSERT(INTT);
        END IF;
```

```
$ Now propagate attributes to the nodes of INTT

        (FORALL U IN NODES)
            SOLN(U) := AUX_F(J) .OF SOLN1;
        END FORALL U;

    END FOR;

    RETURN;
    END PROC FWD_PROPAGATE_IN;
```

```
    PROC INTRAPROC_FWD_ANALYSIS(P, RW F, WR SOLN, ID_PRM, ZERO_PRM,
                                MEET_FLAG_PRM, MOVE_CODE,
                                RW EXPOSED, WR INSERT);

$ This is the master routine to perform a specific data flow
$ analysis intraproceduraLly for a given routine P, within which
$ local variables are analyzed.

$ For more details and comments and description of parameters see the
$ corresponding interprocedural analyser.

    ID := ID_PRM;
    MEET_FLAG := MEET_FLAG_PRM;

    AUX_F := {};

    FLAG := INTRAPROC_FWD_ELIMINATE(P,AUX_F,F,'INTRA');
$ The return value of that procedure is not used in this case

    IF MOVE_CODE THEN
        INSERT := {};
        PROPAGATE_EXPOSED(P, F, AUX_F, EXPOSED, INSERT);
    END IF;

    SOLN := {};
    FWD_PROPAGATE_IN(P, F, AUX_F, SOLN, ZERO_PRM, MOVE_CODE, INSERT);

    RETURN;
    END PROC INTRAPROC_FWD_ANALYSIS;
```

```
PROC INTERPROC_BACK_ANALYSIS(RW F, WR SOLN, ID_PRM, ZERO_PRM,
                             MEET_FLAG_PRM);

$ This is the master routine for performing a specific
$ interprocedural backward data  flow analysis. See the
$ corresponding forward routine for general comments and
$ description of parameters. Here we comment only on differences
$ between the forward and backward algorithms, which are as follows:

$ a. Functional composition must be computed in reverse order.

$ b. The auxiliary maps used in backward analysis are defined as
$ follows: Let I be an interval, J a node in I and V a node outside
$ I which is a successor of a node in I. Then AUX_F([J, V])
$ is defined to be the propagation effect experienced as control
$ advances from the start of U, through  I, to the start of V.

$ To compute this map requires iterating through I in reverse
$ interval order three times (if I is proper) or till convergence
$ otherwise.

$ Since the outermost interval of a procedure P has no
$ successors, we regard the blocks REXIT(P) and RSTOP(P)
$ as its successors, 'hidden' inside that interval.
$ this is needed to enable us to record the effect
$ of the flow through the outermost interval in a manner
$ similar to that used for inner intervals.

$ c. In backward analysis we perform an extra step after the
$ elimination phase. In this step we compute an additional set
$ 'FEXIT' of auxiliary maps. For each node U in P, FEXIT(U)
$ represents the propagation effect of the flow from the start
$ of U to the return block of P, combined with that of flow from
$ the start of U to the stop block of P.

$ d. In our backward analysis code motion issues are completely
$ ignored.

$ e. The technical problem concerning endless loops discussed in
$ SECTION 5 is assumed to be resolved by preliminary processing
$ of the flow graph, in the manner suggested there.

$ Transfer constant parmeters to globals

    ID := ID_PRM;
    ZERO := ZERO_PRM;
    MEET_FLAG := MEET_FLAG_PRM;
```

```
$ This master procedure consists of the following four phases:

$ Interprocedural elimination phase
    AUX_F := INTERPROC_BACK_ELIMINATE(F);

$ Compute auxiliary FEXIT maps.
    FEXIT := {};
    (FORALL P IN ROUTS)
        INTRA_AUX_ELIMINATE(P, F, AUX_F, FEXIT);
    END FORALL P;

$ Find data at procedure exits
    EX_INF := EXIT_INFO(F, AUX_F, FEXIT);

$ Final propagation phase
    SOLN := {};      $ Initialize the solution
    (FORALL P IN ROUTS)
        BACK_PROPAGATE_IN(P, FEXIT, SOLN, EX_INF(P));
    END FORALL;

    RETURN;

    END PROC INTERPROC_BACK_ANALYSIS;
```

```
    PROC INTERPROC_BACK_ELIMINATE(RW F);

 $ This is the driver routine for the interprocedural first
 $ inner-to-outer interval pass. Procedures are analyzed in
 $ the following order: We process the strongly connected
 $ components of the call graph in their postorder; then, for each
 $ such component, we iterate through its procedures in their
 $ postorder, no more than 2*D+1 times, where D is the loop-
 $ interconnectedness parameter of the component.

    AUX_F := {};    $ Initialize auxiliary maps
    F_P := {};  $ Propagation effect thru procedures

$ Iterate through the S.C.C.s of CGRAPH
    (FOR I := #CG_SCCS, #CG_SCCS-1 ... 1)

        SCC := CG_SCCS(I);    $ Get a S.C.C.
        SCC_PROCS := SCC_NODES(SCC);  $ Procs in that S.C.C.
        FLOW_FLAG := 'FIRST_INTER'; $ First processing of SCC

        (FOR J := 1 ... 2 * SCC_D(SCC) + 1 UNTIL PROC_CONVERGE)

            PROC_CONVERGE := TRUE;

            (FOR K := #SCC_PROCS, #SCC_PROCS-1 ... 1)

                P := SCC_PROCS(K);
                PROC_CONVERGE :=
                    INTRAPROC_BACK_ELIMINATE(P,AUX_F,F,F_P,FLOW_FLAG)
                        AND PROC_CONVERGE;
 $ This routine analyzes P; its fifth parameter indicates whether
 $ the analysis is first-time interprocedural, second-time
 $ interprocedural or intraprocedural; it returns a flag to
 $ indicate whether information has stabilized in P.

            END FOR K;

            FLOW_FLAG := 'SECOND_INTER'; $ Additional passes thru SCC

        END FOR J;

    END FOR I;

    RETURN AUX_F;

    END PROC INTERPROC_BACK_ELIMINATE;
```

```
    IF FLOW_FLAG /= 'INTRA' THEN
$ Interprocedural analysis.

        (FORALL C IN CALLSIN(P))

            V := CESSOR(C);     $ The block following the call
            P1 := CALLPROC(C);  $ C calls P1

$ (Note that if this routine is modified to include
$ parameter-passing assignments as part of call blocks, in the
$ manner mentioned above, then one might manipulate F_P(P1),
$ the local effect of executing P1, to get F(C,V)), rather than
$ just assign F_P(P1) to F(C, V)), as is done below).

            IF F(C, V) /= F_P(P1) THEN

$ Update flow function for call
                F(C, V) := IF F_P(P1) = OM THEN FOM
                           ELSE F_P(P1) END;

$ Interval containing call must be processed
                NEED_PROCESS WITH INTOF(C);

            END IF;

        END FORALL C;

$ If no intervals need be processed then information has
$ stabilized and no re-processing of P need be done.
        IF NEED_PROCESS = {} THEN RETURN TRUE; END;

    END IF;

    P_INTS := INTS(P); $ Intervals of P in reverse preorder
    OUTINT := P_INTS(#P_INTS);  $ Outermost interval
    VEDGES(OUTINT) := {REXIT(P)};  $ 'Successors' of OUTINT
    IF (SP := RSTOP(P)) /= OM THEN
        VEDGES(OUTINT) WITH SP;
    END IF;

    (FORALL INTT := P_INTS(I) ST INTT IN NEED_PROCESS)

        NEED_PROCESS WITH INTOF(INTT);  $ Process containing interval
        NODES := INT_NODES(INTT); $ Nodes of INTT in interval order
        HEAD := NODES(1);  $ Interval head

$ Get successor nodes
        CESORS := VEDGES(INTT);
```

```
$ Initialize AUX_F for successor nodes. This trick simplifies
$ subsequent code considerably.

        (FORALL V IN CESORS)
            AUX_F([V, V]) := ID;
        END FORALL V;


$ Three cases are now possible:
$ (1) INTT is proper, but not outermost; then iterate three times.
$ (2) INTT is proper, and is outermost; then iterate once.
$ (3) INTT is improper; iterate indefinitely (1 + 2*number of
$     nodes is an adequate upper bound) until convergence. (Here,
$     again, a better bound can be used; cf. SECTION 6).

        CONV_CONTROL := INTT NOTIN PROPER_INTS;
$ Test for convergence only for improper intervals.

        N_ITER :=  $ Maximal number of iterations thru nodes of INTT
            IF INTT NOTIN PROPER_INTS THEN 1 + 2 * #NODES
            ELSEIF INTT = OUTINT THEN 1 ELSE 3 END;

        (FORALL ND IN NODES, V IN CESORS ST ND /= V)
            AUX_F([ND, V]) := FOM;      $ Initialize auxiliary maps
        END FORALL;

$ Iterate through nodes of INTT.
        (FOR D := 1 ... N_ITER UNTIL CONVRGD)

            CONVRGD := CONV_CONTROL;

$ Iterate thru nodes of INTT in reverse interval order.
            (FOR J := #NODES, #NODES-1 ... 1)

                ND := NODES(J);

                (FORALL V IN CESORS ST V /= ND)

$ Since the 'successors' of the outermost interval are nodes
$ of that interval, we may have ND = V. In this case
$ it would be erroneouss to compute AUX_F([ND, V]) (which has
$ already been set to ID) using the following 'propagation
$ from successors' formula, so we just skip such cases.

                    FTEMP := .MEETJOIN/
                        {F([V], SND]) .COMP AUX_F([SND], V]) :
                            SND IN CESSOR{ND} ST
                            INTOF(SND) = INTT OR SND = V};
```

```
$ Note that flow graph edges (virtual or real) are either
$ edges within an interval, linking two nodes in the same
$ interval, or edges going out of an interval, or edges going
$ into an interval (these last edges are edges from (a target
$ block of) an interval to its head. It is this third kind of
$ edge that we wish to avoid propagating through in the above
$ formula. 'INTOF(SND) = INTT' tests for internal edges
$ and 'SND = V' tests for outgoing edges whose target is V.

                    IF FTEMP = OM THEN FTEMP := FOM; END;
                    CONVRGD := CONVRGD AND
                            (FTEMP = AUX_F([N), V]));
                    AUX_F([ND, V]) := FTEMP;

                END FORALL V;

            END FOR J;

        END FOR D;
$ (Note that no special handling of INTT's head is required.)

$ Except for the outermost interval, compute F([INTT, V]), where V
$ is a successor of some node in INTT.

        IF INTT /= OUTINT THEN
$ F([INTT, V]) is trivially calculated in this case; we also
$ remove the dummy AUX_F([V, V]) entries.

            (FORALL V IN CESORS)
                FTEMP := F([INTT,HEAD]) .COMP AUX_F([HEAD,V]);
                F([INTT,V]) := FTEMP;
                AUX_F([V, V]) := OM; $ To remove this entry from AUX_F
            END FORALL V;
        END IF;

    END FORALL INTT;

$ Compute F_P(P)
    F_P(P) := AUX_F([HEAD, REXIT(P)]); $ HEAD = RENTRY(p)

    IF RSTOP(P) /= OM THEN   $ If P contains a stop block, calculate
                $ propogation effect to that block and combine
                $ it with 'normal' flow effect.

        FZERO := AUX_F([HEA), RSTOP(P)]) .OF ZERO;
        F_P(P) := F_P(P) .MEETJOIN [FZERO, FZERO];
$ Note that a constant function C is represented by [C, C]

    END IF;

    VEDGES(OUTINT) := {};   $ Remove artificial edges added earlier
    RETURN FALSE;   $ To indicate no convergence

    END PROC INTRAPROC_BACK_ELIMINATE;
```

```
    PROC INTRA_AUX_ELIMINATE(P, F, AUX_F, RW FEXIT);

$ This procedure performs an additional intraprocedural
$ elimination, during which we compute, for each node N in P,
$ a map FEXIT(N) representing the effect of flow from the
$ start of N up to an exit of P.

    P_INTS := INTS(P);
    OUTINT := P_INTS(#P_INTS);
    EP := REXIT(P);
    SP := RSTOP(P);

$ First process nodes of OUTINT

    OUTNODES := INT_NODES(OUTINT);

    (FORALL ND := OUTNODES(I))

        FEXIT(ND) := AUX_F([ND, EP]); $ Get the effect of flow to EP

        IF SP /= OM THEN  $ If there is also a stop

            FZERO := AUX_F([ND, SP]) .OF ZERO;
            FTEMP := IF FZERO = XOM THEN FOM ELSE [FZERO,FZERO] END;
            FEXIT(ND) := FEXIT(ND) .MEETJOIN FTEMP;

        END IF;

    END FORALL ND;

$ Next process all remaining intervals in outer-to-inner order
    (FOR J := #P_INTS-1, #P_INTS-2 ... 1)

        INTT := P_INTS(J);
        CESORS := VEDGES(INTT);

        NODES := INT_NODES(INTT);
        (FORALL ND := NODES(K))

            FEXIT(ND) := .MEETJOIN /
                    {AUX_F([ND, V]) .COMP FEXIT(V) : V IN CESORS};

        END FORALL ND;

    END FOR J;

    RETURN;

    END PROC INTRA_AUX_ELIMINATE;
```

```
    PROC EXIT_INFO(F, AUX_F, FEXIT);

$ This function calculates and returns a mapping which sends
$ each procedure P into the flow information available at exit
$ from P. It is called (only in the interprocedural case) just
$ before we begin the final outer-to-inner propagation phase.

$ First we construct a map 'CGF' assigning, to each edge (P, Q)
$ of the call graph, a data-propagation map describing the
$ propagation effect as control returns from the exit of Q to P
$ after any call in P to Q, and then advances to the exit of P.

    CGF := {};
    (FORALL [P,Q] IN CGRAPH) CGF([P,Q]) := FOR; END;

    (FORALL Q := CALLPROC(C))  $ For all calls within all procedures

        P := ROUTOF(C);   $ [P, Q] is an edge of the call graph
        C1 := CESSOR(C);  $ C1 is the block immediately after C
        CGF([P, Q]) := CGF([P, Q]) .MEETJOIN FEXIT(C1);

$ Note that since we are dealing with a backward analysis, we
$ want to propagate data from the exit of the calling procedure P
$ to the exit of the called procedure Q. This direction of
$ propagation, however, makes our problem a forward problem
$ for the call graph.

    END FORALL Q;

$ Next we iterate through the call graph in 'invocation order', i.e.
$ process the strongly connected components in reverse postorder
$ and the set of procedures within each strongly connected
$ component in reverse postorder also.

    EX_INF := { [P, XOM] : P IN ROUTS };  $ Initialize solution
    EX_INF(SYM_MAIN) := ZER;
    CGRINV := {[P, Q] : [Q, P] IN CGRAPH};

    (FOR I := 2 ... #CG_SCCS)   $ Pick S.C.C.'s in reverse postorder

$ Note that we assume here that the main program is non-recursive,
$ so that the first strongly-connected component of the call
$ graph consists of the main program only. Thus we need not process
$ it, for the exit value of the main program is already assumed
$ known.

        SCC := CG_SCCS(I);
        SCC_PROCS := SCC_NODES(SCC); $ Procs in SCC in rev. Postorder
```

```
        (FOR N := 1 ... SCC_D(SCC) + 1 UNTIL CONVRGD)

            CONVRGD := TRUE;

            (FORALL P := SCC_PROCS(K))

                TEMP := .MJV/ (CGF(CQ, P]) .OF EX_INF(Q) :
                        Q IN CGRINV(P)};

$ Test for convergence
                CONVRGD := CONVRGD AND (TEMP = EX_INF(P));
                EX_INF(P) := TEMP;

            END FORALL P;

        END FOR N;

    END FOR I;

    RETURN EX_INF;

    END PROC EXIT_INFO;
```

```
    PROC BACK_PROPAGATE_IN(P, FEXIT, RW SOLN, EX_VAL);

$ This procedure performs outer-to-inner back propagation for
$ a routine P, using the 'FEXIT' information. EX_VAL is the flow
$ information assumed (or known) at the procedure return block,
$ where 'ZERO' is always assumed at the stop block of P (but this
$ assumption has already been used in calculating the FEXIT maps).

    (FORALL INTT IN INTS(P), J IN INT_NODES(INTT))

        SOLN(U) := FEXIT(U) .OF EX_VAL;

    END FORALL;


    RETURN;

    END PROC BACK_PROPAGATE_IN;
```

```
    PROC INTRAPROC_BACK_ANALYSIS(P, RW F, WR SOLN, ID_PRM, ZERO_PRM,
                                 MEET_FLAG_PRM);

$ This is the master routine to perform a specific backward data flow
$ analysis intraprocedurally for a routine P whose local
$ variables are to be analyzed. For more details, comments, and
$ description of parameters see the corresponding interprocedural
$ analyser.

    ID := ID_PRM;
    ZERO := ZERO_PRM;
    MEET_FLAG := MEET_FLAG_PRM;

    AUX_F := F_P := {};

    FLAG := INTRAPROC_BACK_ELIMINATE(P, AUX_F, F, F_P, 'INTRA');
$ Flag is not used in this case

    FEXIT := {};
    INTRA_AUX_ELIMINATE(P, F, AUX_F, FEXIT);

    SOLN := {};
    BACK_PROPAGATE_IN(P, FEXIT, SOLN, ZERO);

$ Note that in the intraprocedural case the last two procedures
$ can be combined to form a single procedure almost identical
$ with 'INTRA_AUX_ELIMINATE', except that this procedure computes the
$ 'SOLN' map directly instead of the 'FEXIT' maps.

    RETURN;

    END PROC INTRAPROC_BACK_ANALYSIS;
```

```
$ Here are the operators which manipulate the data propagation maps
$ and data states.

    OP .COMP(G, F);    $ Functional composition G of F

    RETURN
        IF F = FOM OR G = FOM THEN FOM
        ELSE [F(1) * G(1) + G(2), F(2) * G(1) + G(2)]
        END;
    END OP .COMP;


    OP .MEETJOIN(G, F);    $ Functional meet or join
    RETURN
        IF F = FOM THEN G
        ELSEIF G = FOM THEN F
        ELSEIF MEET_FLAG THEN [F(1) * G(1), F(2) * G(2)]
        ELSE [F(1) + G(1), F(2) + G(2)]
        END;
    END OP .MEETJOIN;


    OP .MJV(X, Y);    $ Meet or join of lattice elements
    RETURN
        IF X = XOM THEN Y
        ELSEIF Y = XOM THEN X
        ELSEIF MEET_FLAG THEN X * Y
        ELSE X + Y
        END;
    END OP .MJV;


    OP .OF(F, X);    $ Functional application
    RETURN IF X = XOM OR F = FOM THEN XOM
        ELSE F(1)*X + F(2)
        END;
    END OP .OF;



    END MODULE SETL_OPTIMIZER - DATAFLOW_SOLVER;
```

NYU NSO-17 c.2
Schwartz
AUTHOR
A design for optimizations
TITLE
of the bitvectoring class.

DATE DUE	BORROWER'S NAME